Improving the Efficiency of Dynamic Malware Analysis

Ulrich Bayer
Technical University Vienna
Treitlstrasse 1
1040 Vienna, Austria
+43 1 5880118314
ulli@seclab.tuwien.ac.at

Engin Kirda
Institute Eurecom
2229, Route des Cretes
F-06560 Sophia-Antipolis
+33 4 9300 8247
kirda@eurecom.fr

Christopher Kruegel
University of California
Santa Barbara
CA 93106-5110, USA
+1 (805) 893-6198
chris@cs.ucsb.edu

ABSTRACT

Each day, security companies see themselves confronted with thousands of new malware programs. To cope with these large quantities, researchers and practitioners alike have developed dynamic malware analysis systems. These systems automatically execute a program in a controlled environment and produce a report describing the program's behavior. During the last three years, the number of malware programs appearing each day has increased by a factor of ten, and this number is expected to continue to grow. To keep pace with these developments without causing even more hardware costs for operating dynamic analysis systems, we have developed a technique that drastically reduces the overall analysis time. Our solution is based on the insight that the huge number of new malicious files is due to mutations of only a few malware programs. To save analysis time, we suggest a technique that avoids performing a full analysis of the same polymorphic file multiple times. In an experiment conducted on a set of 10,922 randomly chosen executable files, our prototype implementation was able to avoid a full dynamic analysis in 25.25 percent of the cases.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*

General Terms

Security

Keywords

dynamic analysis, malware analysis

1. INTRODUCTION

The root cause of many criminal activities on the Internet are malicious programs. Trojans, viruses, bots, etc. give miscreants a wide range of possibilities for conning unsuspecting Internet users. For this reason, we see a huge number of new malware programs appearing each day. This number has grown dramatically over the last few years, and it will continue to grow in all likelihood. At the time of writing this paper, one has to assume that around 35,000 new malicious binaries appear each day. Obviously, A/V companies cannot analyze such a high number of files manually. They need automated tools for verifying whether all of the suspicious files they receive are, in fact, malicious or not. Because of the limits of static analysis [25], this prompted researchers and practitioners to develop automated, dynamic malware analysis systems.

Automated, dynamic malware analysis systems work by running a binary in a safe environment, monitoring the program's execution and generating an analysis report summarizing the behavior of the program. These analysis reports typically cover file activities (e.g., what files were created), Windows registry activities (e.g., what registry values were set), network activities (e.g., what files were downloaded, what exploit were sent over the cable), Windows service activities (e.g., what services were installed) and process activities (e.g., what processes were terminated). Several of them are publicly available on the Internet (Anubis [1, 12], CWSandbox [4], Joebox [8], Norman Sandbox [9], ThreatExpert [10]) but many similar internal systems exist behind the closed doors of A/V companies. The main thing to note about dynamic analysis systems is that they are indeed executing the binary for a limited amount of time. Since, typically, malicious programs do not reveal their behavior when only executed for several seconds, dynamic systems are required to monitor the binary's execution for a longer time. This is why dynamic analysis is resource-intensive in terms of necessary hardware and time. Moreover, the sheer number of malware programs appearing each day became high enough to not only challenge manual analysis but also automated, dynamic malware analysis. One needs costly server farms running the dynamic analysis systems to cope with the ever-increasing load (i.e., amount of binaries to be analyzed).

In this paper, we present a novel and practical approach for improving the efficiency of dynamic malware analysis systems. Our approach is based on the insight that the huge number of new malicious files appearing each day is due to mutations of only a few malware programs [18]. More precisely, malware authors write programs that reproduce polymorphically [26] or employ runtime packing algorithms to create new malware instances that differ on the file level, but exhibit the same behavior. We propose a system that avoids analyzing malware binaries that merely constitute slightly mutated instances of already analyzed polymorphic mal-

SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

ware. To detect polymorphic binaries, we have extended our dynamic analysis system to check—after executing the malware program for only a short time—whether our database of existing analysis reports contains a behaviorally almost identical (for the time frame in question) analysis report. If this is the case, we stop the analysis process and instead, return the existing analysis result.

The contributions of our paper are as follows:

- We propose an approach that drastically reduces the amount of time required for analyzing a set of malware programs. To achieve this, we avoid analyzing the same polymorphic program multiple times. For detecting that a program is a polymorphic variation of an already analyzed binary, we dynamically analyze it for a short period of time. In a next step, we search the behaviorally nearest program. If such a program is similar enough (with respect to a specified threshold), we stop the currently ongoing analysis and instead return the existing analysis result.

- We present experimental evidence that demonstrates that our approach is feasible and usable in practice.

- We have designed an algorithm that is efficient and scalable. We find a program's behaviorally nearest neighbor without having to perform $n-1$ comparisons.

2. BACKGROUND: ANALYSIS TIME

A dynamic malware analysis system faces the problem that it has to analyze as many suspicious binaries as possible within a limited time frame and a limited amount of computing resources available. At the same time, it still has to provide meaningful analysis reports. Clearly, it is necessary that a dynamic analysis system executes and monitors a given binary for a reasonable amount of time to determine the binary's purpose. Traditional systems either analyze a given binary until its execution as well as the execution of all of its children processes ends, or a certain timeout limit has been reached. This timeout is four minutes long in the case of the dynamic analysis system that we are modifying. This means that the execution of a binary under analysis lasts for a maximum of four minutes. The total analysis time for a file, however, might be longer because in most cases, a post-processing step follows the actual execution phase. Our system, for instance, permits the post-processing step to run for a maximum of another four minutes. In case the program exits (or dies because of an error) before the timeout is reached, the analysis will naturally take less time. The assumption behind this modus operandi is that the typical malicious program tries to perform its malicious actions as soon as possible. However, we want to point out that this assumption is not always true and that a longer analysis might be desirable in some situations . For example, a binary could try to sleep for several minutes before it begins its (malicious) work. To allow for a longer analysis (in certain cases or in general), even more computing resources are required. In the following paragraphs, we will describe a technique that reduces the amount of required analysis time. Thus, this technique helps a dynamic analysis system both to analyze more programs in a given period and to analyze programs for a longer amount of time. More formally, this relationship can be described as:

$$OverallAnalysisTime = (|B| \cdot \sum_{b \in B} t_a(b))/I$$

with B being our set of binaries, t_a the analysis time for a single binary and I the number of instances of the analysis system that are running in parallel.

More precisely, the analysis time $t_a(b)$ of a binary b is composed of a *setup time* $t_s(b)$ and a *post-processing time* $t_p(b)$ in addition to the actual time $t_e(b)$ used for executing the binary b in a secure environment. That is:

$$t_a(b) = t_s(b) + t_e(b) + t_p(b)$$

During the setup time, we prepare the analysis environment—possibly by loading a virtual machine and transferring the program into it. In the final post-processing step, we apply all kinds of offline analysis methods to the information gained during the execution of the binary. Tasks performed during the post-processing step range from archiving the analysis result and updating databases to running scripts for analyzing a network traffic dump file.

Note that analysis systems usually treat binaries scheduled for analysis as a mathematical set consisting of unique files. That is, to save analysis time, one avoids analyzing the same file multiple times. Practically, this technique is implemented by computing a hash value for the file before the analysis starts. If a matching analysis result already exists in the report repository, the analysis system can simply return the already existing result to the user.

3. REDUCING THE OVERALL ANALYSIS TIME

Our solution is based on the insight that the large quantity of new malicious files appearing each day is due to mutations of only a few malware programs (e.g., polymorphic reproduction or use of runtime packing algorithms with a random crypt seed resulting in a slightly changed binary). Indeed, we made the experience that, in our system, analysis reports are in many cases almost identical suggesting that we've analyzed a polymorphic malware instance several times. We propose a system that makes use of the fact that we can avoid analyzing the large percentage of incoming malware binaries that merely constitute slightly mutated instances of already analyzed polymorphic malware binaries. This system is the logical extension of the hash-based technique that saves analysis time by not analyzing the same file (identified via its hash value) twice.

To this end, our system checks after running a binary b for only a short amount of time that we call the *checkpoint time* T_c whether the behavior seen in this short time is almost identical to behavior seen in a previous analysis. If this is the case, we stop analyzing the binary b. We return the analysis result of the program that we found to behave almost identically instead. This means that we are able to deliver a full analysis report which covers all of a program's behavior as observed in time $t_e(b)$ for a binary b that we effectively analyzed only for a much shorter period T_c . Of course, this scheme only makes sense if T_c is a lot smaller than $t_e(b)$: $T_c << t_e(b)$. We will discuss the selection of T_c in the evaluation section.

Thus, the analysis time of a pre-empted binary b is given by $t_{pre-empted}(b) = t_s(b) + T_c$. Since in the case of pre-empted binaries we return an already existing analysis result there is no need for a post-processing phase. The time saved by pre-empting a file b is consequently $t_a(b) - t_{pre-empted}(b)$.

3.1 Behavioral Profiles

To determine whether a program's behavior after time T_c corresponds to one that we have already analyzed, we leverage a presentation of a program's behavior that we call *behavioral profile*. We represent a binary's b behavioral profile as $bp(b)$ in this paper. Behavioral profiles were introduced in [13]. A behavioral profile aims to capture a program's behavior at an higher level of abstraction than a raw system call trace while correctly retaining a program's behavioral semantics. Among other things, a behavioral profile relies on information gained from the data tainting system of a dynamic analysis program . This is used, for example, in order to determine whether execution artifacts, such as filenames, registry-key names, etc. depend on randomness and thus change with every execution of the program. Clearly, it is of utter importance to detect randomness when comparing two behavioral profiles. It is known, for example, that the polymorphic Allaple worm scans a randomly chosen IP sub-net for potential victims. Even in the simplistic case of comparing two different executions of the same binary we have to detect that the target IP is randomly chosen for achieving a high similarity score. We refer the reader to [13] for the details of behavioral profiles. For this paper, it is important to know that a behavioral profile is essentially a set (in the mathematical sense) of features where a feature could be a string of the form `file|C:\Windows\test.exe|create` to reflect the event that a file named `C:\Windows\test.exe` was created.

For this paper, it proved useful to extend behavioral profiles with timing information. More concretely, we assigned a timestamp value to each feature representing the feature's first occurrence in an execution trace. This permits us to order features based on their first occurrence. Moreover, it allows for more advanced comparison techniques between behavioral profiles. Please note that a behavioral profile still remains a set of string features. If, for instance, a program creates and deletes a certain file several times, the behavioral profile contains only a single feature representing the file's creation. Its timestamp would equal the time when the program created the file the very first time. This timestamp value specifies the offset to a well defined starting time. We decided to use the time when a program's very first user-mode instruction is being executed as the starting time. This approach is robust against varying durations of the setup phase where among other things we have to load the snapshot and copy the program into the virtual environment.

3.2 Comparison

We consider a program b to be a polymorphic variant of another program a if the distance between their behavioral profiles at time T_c is below a certain distance threshold d. Formally, we demand that $dist(bp(a), bp(b)) < d$. As a distance function we employ the Jaccard distance [21], defined as

$$J(a, b) = 1 - |a \cap b| / |a \cup b|$$

We define two programs a and b as being *behaviorally identical* if $J(bp(a), bp(b)) < d$ is true. In the evaluation section, we are going to discuss the selection process for the distance threshold parameter d. In the ideal case, we would expect to have a distance of 0 between the profiles of two behaviorally identical binaries. Practically, our experiments show that this distance is rarely exactly zero. The reason for that is that our behavioral profile cannot capture all randomized artifacts. Another reason is that frequently the analyzed program is not able to execute the exact same number of system calls during different analysis runs. This is due to OS scheduling decisions, differing server workloads, network connection latencies and similarities.

Extended Jaccard Distance. Although we perform our analysis runs each time in exactly the same configuration, we cannot prevent the existence of a small number of differences in behavioral profiles due to timing issues. Consider, for example, that we have chosen a checkpoint time T_c of 45 seconds and that we have two analysis runs of the same file. Let us furthermore assume that this file is programmed to sleep for 45 seconds and to proceed by creating twelve files. Clearly, it is easily possible that in one execution all twelve files have been created at time T_c while in the other one no files at all have been created. To alleviate this problem, we introduce an extended Jaccard distance J_e that is more robust against the described timing issues.

Without loss of generality we assume that for two behaviorally identical programs a, b the program a has already advanced further in its execution at a certain point in time. Thus, its behavioral profile contains more features. At the same time, b's behavioral profile $bp(b)$ is an approximate subset of $bp(a)$ (since we are assuming that b exhibits the same behavior). We define an approximate subset as a relationship where a large percentage p of the features are the same: $|bp(b) \cap bp(a)| / |bp(b)| >= p$. For our experiments, we have—based on our experience with the reference set—chosen a value of 0.9 for p. We will demonstrate that this selection of p is reasonable and yields good results. This model motivates the following algorithm for computing a more timing resilient distance value:

1. if $bp(b)$ is not an approximate subset of $bp(a)$ we stop and return the normal Jaccard distance $J(bp(a), bp(b))$

2. otherwise we select the feature $f_{highest} \in bp(a) \cap bp(b)$ with the highest timestamp

3. we compute a $bp_{normalized}(a)$ by removing all features from $bp(a)$ with a timestamp higher than $timestamp(f)$

4. return $J(bp_{normalized}(a), bp(b))$ as a result

In our experiments, we compare the results achieved by employing either one of them. The computation of the extended Jaccard distance is more costly than the simple Jaccard distance. Additionally, as we will see in the next section, it lacks some of the properties that make the Jaccard distance so attractive. In particular, techniques exist that allow the efficient search for a behavioral profile's nearest neighbor when the Jaccard distance is used as a distance metric. This is why we use the Jaccard distance as our primary distance metric and resort to the extended Jaccard distance only in a second step when it is computationally more feasible.

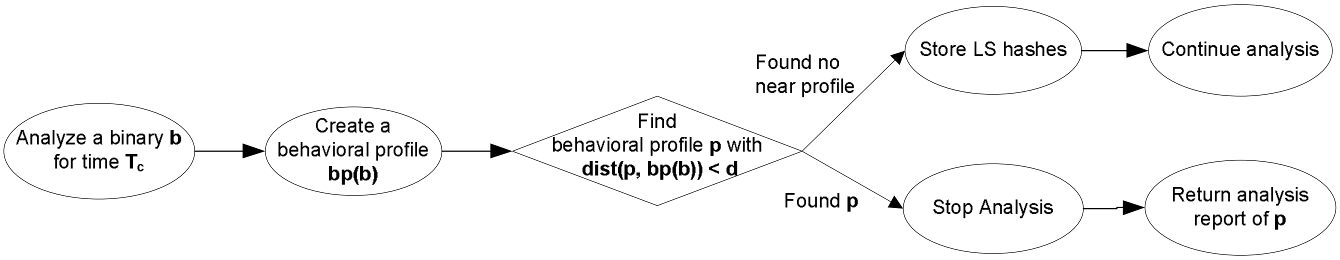
Analyze a binary **b** for time **T**c
Create a behavioral profile **bp(b)**
Find behavioral profile **p** with **dist(p, bp(b)) < d**
Found no near profile
Found **p**
Store LS hashes
Continue analysis
Stop Analysis
Return analysis report of **p**

Figure 1: Overview of our approach for saving analysis time

3.3 Efficient Nearest Neighbor Search

As explained in the preceding paragraphs, we represent a program's behavior in the form of a *behavioral profile* and use the Jaccard distance for determining the dissimilarity between two behavioral profiles. In this subsection, we will describe how to efficiently find for a program b at time T_c, an almost identical program if such a program exists (i.e., was already analyzed). The naive solution, a *linear search*, would be to compare $bp(b)$ with all existing behavioral profiles. This solution has a runtime complexity of $O(n * d)$ where n is the number of behavioral profiles in our database and d is the number of features present in the union of all behavioral profiles. In addition to the runtime costs, we would need to keep all behavioral profiles in main memory to allow for a efficient comparison. This is clearly not very scalable.

A more efficient technique for finding the nearest behavioral profile is *Locality Sensitive Hashing* (LSH) [20]. LSH provides an efficient (sublinear) solution to the approximate nearest neighbor problem (ϵ-NNS). LSH was already successfully leveraged for developing a scalable, malware clustering system [13]. In the following, we will shortly describe LSH to the extent necessary for understanding this paper. The idea behind LSH is to map similar points (in our case behavioral profiles) with high probability to the same hash value. 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. Informally, this means that the more similar two behavioral profiles are, the higher the probability that the hash function will map them to the same value. 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 [14]. 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 $h(x) = c_1x+c_2 \mod P$ are used instead [19], with P a prime number larger than the total number of features in F.

Given the distance threshold d, we choose a suitable number k of hash functions in each LS hash, and moreover the number of iterations l. Thus, the parameters l and k permit us to control the LS hashing such that it yields good results with respect to our distance threshold. Formally, the collision probability of two behavioral profiles is computed as $Pr[collision(a, b)] = 1 - (1 - (sim(a, b)^k))^l$

3.4 The Analysis Process

In this subsection, we explain the steps necessary to integrate our approach into the traditional analysis work flow. First, we we assume that the results of completed analysis runs are being stored. Second, the LSH configuration consisting of $l * k$ hash functions $h_{1,1}, ..., h_{k,l}$ has to be selected once and stored persistently for later use. Third, LS hashes for completed analysis runs need to be stored persistently in a hash database.

Figure 1 gives an overview of the entire analysis process. After analyzing a binary b for T_c seconds, we create a behavioral profile $bp(b)$. This profile captures the program's behavior until time T_c. In a next step, we employ LS hashing to find the set of candidate near behavioral profiles N. To this end, we first initialize the set N to the empty set. Then, this search is performed in l iterations with each iteration consisting of the following steps:

1. We load the k hash functions $h_1, ..., h_k$ for iteration l from our persistent storage
2. We compute the LS hash for binary b: $lsh(b) = h_1(b), ..., h_k(b)$
3. We check whether $lsh(b)$ exists in our database of hashes and add all behavioral profiles with identical LS hashes to the set of candidate neighbors N

Since the set of candidate neighbors N might contain false positives due to the probabilistic nature of LS hashing, we compute all the distances $J(b, n)$ for all $n \in N$. In the evaluation section, we will demonstrate that results of replacing J with J_e for this step. We keep the nearest behavioral profile n if one exists with a distance smaller than our chosen threshold. In case we found a behaviorally identical profile, we stop the currently ongoing analysis and return the analysis report of profile n. Otherwise, we store the l LS hashes that we calculated before in our hash database and let the dynamic analysis continue.

4. EVALUATION

To verify the correctness and efficiency of our approach, we have implemented a prototype system. We will first shortly describe our prototype implementation and then discuss the experiments that we conducted, and the results obtained.

4.1 Prototype Implementation

For testing our approach, we modified an existing dynamic analysis system [12]. In the following, we summarize the most important changes.

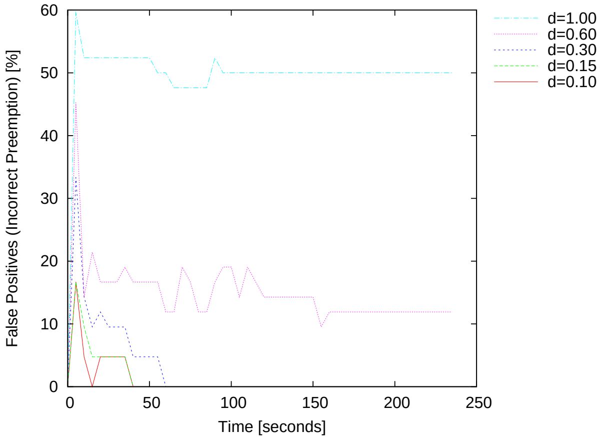

Figure 2: False Positives

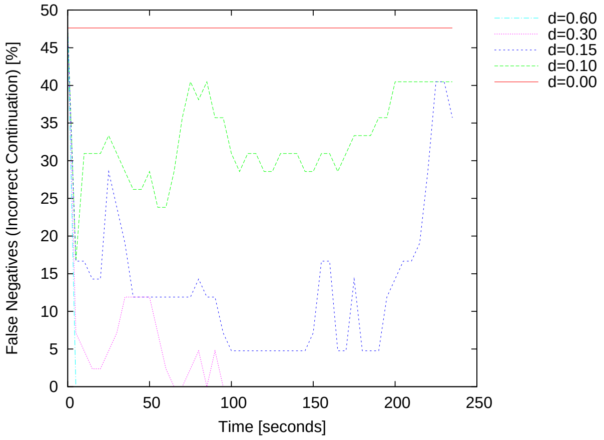

Figure 3: False Negatives

On-the-fly generation of the behavioral profile. First, we had to modify the analysis system so that behavioral profiles are built incrementally while the analysis of a program progresses. Each invocation or return of a system call triggers the update of our behavioral profile. Consequently, it is possible to create a behavioral profile at each point during the analysis of a program. Special handling was necessary to account for network actions. Since the existing network analysis examines the raw network traces, we have to execute this network analysis script (which in turn parses the network traffic dump file) each time a behavioral profile is generated.

Timestamps. The generation of behavioral profiles was modified to include a timestamp for each feature. The timestamp indicates the time of a feature's first occurrence.

LSH. For performance reasons, the LSH computation code is written in C++. To interface this code with the rest of the analysis scripts, which are written in Python, we wrapped the C++ code in a Python module (with the help of Boost.Python [3]). The LS hashes for a profile are stored in a relational DBMS (MySQL). Searching for LS hashes and adding new hashes is performed via regular SQL queries.

Mapping feature strings to integer values. It would be inefficient to perform all of our distance and LSH computations directly with behavioral profiles in the form of sets of (feature) strings. Instead, we map each feature string to a unique integer value with the help of a table in our relational DBMS. Currently, we store feature ids as 32-bit numbers.

LSH configuration. We decided to store the LSH configuration in the relational DB as well. This permits each analysis run to rebuild the identical $k * l$ hash functions $h_{1,1}, ..., h_{k,l}$. The LSH configuration consists of $l*k$ (pseudo-) random numbers c_1, c_2 and the prime number P. It is necessary to select a prime number P that is higher than the number of all features. Since we cannot predict how many features we will have in the future (each analysis adds new features) we chose the largest 32 bit prime number for P.

4.2 Experiment with a Reference Set

To assess the effects of the checkpoint time parameter T_c and the distance threshold d on our algorithm, we chose to compare the outcome of our algorithm under different configurations with a reference set.

Reference Set. In a first step, we manually compiled a set of 20 polymorphic programs and 22 non-polymorphic programs, 42 files in total, that should serve as our reference set. More precisely, we included four different types of malware which are known to be polymorphic and which appeared in the wild during August 2009:

- *Virut*: Virut is a polymorphic file infector [7]. It infects files with an .EXE or .SCR extension by appending a slightly modified copy of itself to the file. We were able to manually verify that the five Virut files in our set were indeed all infections of the same original executable file. That is, the five Virut files contained the same host file and only differed in their last section where the actual (polymorphic) virus code resides.

- *Allaple.1*: Allaple [5] is a polymorphic worm that spreads by exploiting a number of vulnerabilities. Whenever the worm propagates it newly encrypts its code. The result is a copy of the virus that differs at the byte-level from its source.

- *Allaple.2*: This is another variant of Allaple.

- *Trojan-PWS.Win32.LdPinch*: LdPinch [6] is a Trojan that is designed for stealing passwords and mailing them back to the virus author. Unlike viruses and worms Trojans do not replicate. Consequently, someone must have created the different LdPinch files in our set. We discovered that a toolkit for creating Pinch Trojans exists [2] that allows for the easy creation of new Pinch Trojans. We suspect that this or a similar tool was used by the virus author(s) for the semi-automatic creation of new LdPinch files. The toolkit was first detected in the wild in 2008 but newly created variants continue to appear.

Each of the four polymorphic malware programs in the preceding list is represented by five unique binaries in our reference set.

Selecting the checkpoint time and the distance threshold. The time parameter T_c determines after how many seconds we search for the nearest behavioral profile. The distance threshold d specifies the maximum distance that two behavioral profiles are allowed to have in order to be considered as behaviorally identical. To understand the effects of these parameters on our results, we conducted the following experiment.

For all parameters combinations $T_c \in \{1, 2, ..., 240\}$, $d \in \{0.05, 0.1, 0.15, ..., 0.4, 0.5, ..., 1.0\}$ we calculated a full distance matrix based on the Jaccard distance. After choosing the nearest profile for each file, we decided in accordance with the current threshold d whether the analysis of this file is pre-empted or not. Each time, we compared the outcome with the reference set. We measure the success of our algorithm by calculating the number of the *false positives* (i.e., the number of programs that were wrongly determined to be behaviorally identical) and the number of *false negatives* (i.e., the number of programs that we did not correctly identify as being behaviorally identical).

In this experiment, we are mainly interested in the distances between behavioral profiles at specific execution times. This is why, we we do not assume any specific submission order and calculate all possible distances.

Figure 2 shows the percentage of false positives in relation to the checkpoint time. The figure contains five different lines showing the false positive rate for five different distance thresholds. Naturally, a higher distance threshold leads to higher false positive rate. In the extreme case of a distance threshold of $d = 1.00$, all 22 non-polymorphic binaries are wrongly found to be behaviorally identical with one of the four polymorphic malware types. On the other hand, a value of $d = 1.00$ leads to 0 percent of false negatives. Furthermore, we can see that the percentage of false positives diminishes over time. This makes perfectly sense because the longer we execute a program the more characteristic actions we are going to include in its behavioral profile.

We show the false negative rate in Figure 3. In contrast to the false positives, the false negative rate improves when the distance threshold increases. A threshold of $d = 0.30$ paired with a checkpoint time greater than approximately 100 seconds is sufficient to not miss any polymorphic binary. That is, we recognize all polymorphic programs correctly as being polymorphic. In the extreme case of $d = 0.00$, none of our 20 polymorphic programs is correctly recognized as being polymorphic. Moreover, it is interesting to see that a distance threshold of 0.10 still results in quite a high number of false negatives. This indicates that our behavioral profiles still contain more execution-specific artifacts than we desire. It is noteworthy that the false negative rate fluctuates a lot at different checkpoint times in case of tight thresholds, such as 0.10 or 0.15. This is because a longer analysis time increases the number of actions and the number of execution artifacts in a behavioral profile. As a consequence, a longer analysis time can make behavioral profiles drift more apart than would be appropriate.

4.3 Real-World Experiments

After completing our experiments with the reference set, we started to test our algorithm in a real-world setting. To this end, we installed and operated our prototype system inside our dynamic analysis platform for several days. In this period, the system has analyzed a set B of 10,922 unique executable files. For each analysis, we created and stored a behavioral profile including timestamps for all features. In this experiment, we did not stop the analysis of any programs prematurely. Instead, we decided to allow all programs to continue running until the normal timeout is reached. This puts us into a position to review the full behavior of otherwise pre-empted programs and to reason whether our algorithm's decision to prematurely end an analysis run is justified.

While it is straightforward to calculate the total amount of time saved by running our algorithm in this real-world setting, it is inherently more difficult to estimate the number of false positives and false negatives in these results. We developed the following strategy to evaluate how reliable our algorithm is in its decisions to prematurely stop an analysis : For all the programs $b_i \in B$, that, according to our algorithm, have a behaviorally identical program $s_i \in B$ at checkpoint time T_C, we compute the pair-wise Jaccard distances at the time of the traditional analysis end. In other words, we are evaluating how much the behavior of two programs a and s that were found to be behaviorally identical at an earlier time T_c differs after the normal timeout of four minutes. This distance calculation permits us to quantify to what extent our analysis result would have differed in case our algorithm was actively deployed. We are not claiming that the current analysis results, which are delivered after 4 to 8 minutes, are always correct. We are only examining the question to what degree our algorithm, while saving time, returns possibly worse analysis results. This strategy does not allow us to directly measure the false positive rate, but it is suited to give us an estimate of the false positive rate.

Although the size of the real-world set makes an exhaustive evaluation of all possible parameters infeasible, we were able to try out several interesting ones thanks to the fact that we had the full behavioral profile including timestamps available. We show the results of performing five runs of our algorithm on set B with varying parameters for the checkpoint time T_c and the distance threshold d in Table 1. Our initial parameter selection was guided by our experience gained while performing tests on the reference set. Additionally, we were examining the effects of using the extended Jaccard distance.

Configuration	*Pre-empted files*	*Time saved/ pre-emption*	*Total time saved*
45s, 0.12	3,087 (28.26%)	265s	227.2 hours
60s, 0.12	2,747 (25.15%)	250s	190.8 hours
60s, 0.12, J_e	3,659 (33.5%)	250s	284.1 hours
60s, 0.08	1,653 (15.13%)	250s	114.8 hours
60s, 0.08, J_e	2,539 (23.24%)	250s	176.2 hours

Table 1: Results of testing our approach in different configurations on a set of 10,922 binaries

When testing a new configuration, of course, no LS hashes exist in the beginning. Clearly, for the very first program, we cannot possibly find a nearest neighbor. During each run of our algorithm, we performed the steps detailed in section 3.4. This means that we iterated (in the same order that the files were originally analyzed) over the set of files in B: For each $b \in B$, we searched the nearest behavioral profile $s_i \in B$. This search process consists of the LS hashing step that efficiently calculates a set of candidate near profiles and a subsequent traditional comparison with all the candidate near profiles to eliminate false positives. The number

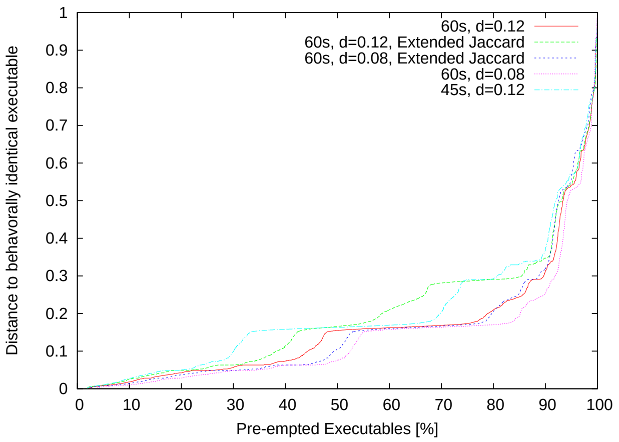

Figure 4: CDF in [%] of distances $J(b_i, s_i)$ at time t_e

of traditional comparisons averaged to 1.2820 during all our runs. For practical reasons, we conducted all our algorithm runs with a value of $l = 140$ and $k = 25$. We selected these values to get good results for distances of 0.15 and smaller. More precisely, these parameters cause behavioral profiles with a distance of 0.15 to collide with a probability of 0.912. The chosen value for k and l allowed us to test our algorithm reliably with all threshold distances smaller than 0.15. For thresholds < 0.15 we simply adapted our traditional comparison function, which checks all candidate near profiles emitted by the LSH step, accordingly. Furthermore, for the runs in Table 1 having J_e listed in their configuration, we performed the false positive removal by computing the extended Jaccard distance.

As stated before, for all binaries that have a behaviorally identical program at time T_c, we recomputed the distance at time t_e. Figure 4 shows the distribution of these distances as a CDF. The x-axis of the diagram details the percentage of behaviorally identical programs while the y-axis specifies the distance between two programs at time t_e (i.e., after four minutes). One can see that in most parameter configurations around 90% of all pre-empted executables have a distance $<$ 0.3 after executing for four minutes. We saw in Figure 3 that executions of the same malware program can easily have a distance of 0.3. In fact, a threshold below 0.3 leads to a number of false positives for checkpoint times below 100 seconds. At the same time, a distance threshold of 0.3 causes no false positives, as can be seen in Figure 2. This is why these results are absolutely encouraging. They demonstrate that our algorithm works well with only a small number of serious distance deviations.

Looking at Figure 4 makes it clear that we get better results in the runs where we chose a checkpoint time of 60 seconds as opposed to 45 seconds. Moreover, we see that the extended Jaccard distance does not perform necessarily better. It is also quite intuitive that demanding a smaller distance threshold at time T_c results in smaller distances at time t_e. At the same time, the effectiveness (i.e., number of pre-empted files) of our algorithm decreases with tighter distance thresholds as can be seen in Table 1. Thus, there is no single correct value for the checkpoint time and the distance threshold. When selecting these parameters, one has to take the requirements of the application into consideration. For our purposes, we believe that a checkpoint time of 60 seconds and and a distance threshold of 0.12 strike a good balance between reliability of the analysis results and efficiency of the algorithm.

To calculate the time saved by pre-empting an analysis run, we make use of average values for $t_a(b)$ and for $t_{pre-empted}(b)$. In case of our dynamic analysis system, the average analysis time for a program including the setup-time and post-processing amounts to 334 seconds. For pre-empted analysis runs, we have to add on average 24 seconds to the checkpoint time to account for the setup-time. As a consequence, we save on average 265 seconds with a checkpoint time of 45 seconds and 250 seconds when the checkpoint time is 60 seconds. A configuration of $T_c = 60s$ and $d = 0.12$ saves in total 190.8 hours of analysis time. In this time, we can perform 2,056 additional, full analysis runs on average.

5. LIMITATIONS

It is obviously possible that a malicious adversary crafts two files that appear to be behaviorally identical at checkpoint time T_C but change their behavior afterward. In such a case, our algorithm would wrongly pre-empt the analysis of one file. There is no easy defense against this kind of attack since this is an intrinsic problem of dynamic analysis systems. A dynamic analysis system is evadable by programs that do not reveal their true behavior during the short period where their behavior is monitored. While it is possible to defend against specific attacks (such as sleep operations), it is more difficult to find solutions in the general case.

6. RELATED WORK

Dynamic analysis system are not the only means to analyze malicious binaries. System based on static analysis (e.g., [15, 16, 17, 22]) also exist. They are less popular though because malware is usually well-protected against static techniques. In particular, today's malware programs leverage code obfuscation [24], code encryption and runtime packing [18] to make dis-assembly difficult. Since all static techniques that are more sophisticated rely on disassembling the binary in a first step these techniques suffer from the fact that they cannot analyze the majority of malicious binaries. The biggest advantage of static techniques is their potential to reason about all possible execution paths of a (malware) program while dynamic analysis is limited to a single execution path. In theory, also static techniques could profit from the fact that a large portion of today's malicious program landscape is composed of file-level variations of a small number of malware programs. Analogous to our technique for dynamic analysis systems an expensive static analysis run could be avoided if it's possible to prematurely detect that an analysis of this malware program already exists.

Another related technique is behavior-based malware clustering [11, 13, 23] . Both clustering systems, as well, as our algorithm for detecting already analyzed programs (albeit different on the file level) have to compare different execution traces and define a notion of similarity. While clustering aims to find groups of behaviorally similar programs we are only interested in finding a program's nearest neighbor. Finding a program's nearest neighbor is a step necessary to reach our goal of not analyzing binaries that have already been analyzed. We are performing this search based on monitoring the behavior for a more limited amount of time. Clustering systems, on the other hand, work of course on the whole execution trace of a binary.

7. CONCLUSIONS

In this paper, we propose a novel approach for making the dynamic analysis of malicious programs more efficient. It drastically reduces the amount of time necessary for analyzing a set of malicious program. Our approach makes use of the fact that the huge number of new malware programs appearing each day is due to mutations of only a few malware programs. Therefore, we suggest a technique that avoids fully analyzing a program again if we have already analyzed this program (albeit different on the file level) once. We detect that a program is a polymorphic variation of an already analyzed binary by executing it for a short period of time. In a next step, we check whether the behavior seen in this period is almost identical to an already analyzed binary. If this is the case, we stop the currently ongoing analysis and instead return the existing analysis result.

We have empirically demonstrated that this technique works well in practice and that it is efficient. By leveraging locality sensitive hashing we avoid performing $n-1$ comparisons for determining whether an almost identical program has already been analyzed. Moreover, our experiments show that for a set of 10,922 randomly chosen executable files, we were able to avoid the full analysis of 2,747 files (25.25%). This equals 190.8 hours of saved analysis time. In the future, we plan to actively use this technique in our dynamic analysis system because it helps us to analyze more of today's malicious programs.

8. ACKNOWLEDGMENTS

This work has been supported by the European Commission through project FP7-ICT-216026-WOMBAT and FP7-ICT-216331-FORWARD, by FIT-IT through the Pathfinder project, by FWF through the Web-Defense project (No. P18764) and by Secure Business Austria.

9. REFERENCES

[1] ANUBIS. `http://anubis.iseclab.org`, 2009.

[2] Article about Pinch in the Washington Post. `http://blog.washingtonpost.com/securityfix/2007/11/new_malware_defeats_sitekey_te.html?nav=rss_blog`, 2009.

[3] Boost.Python C++ library. `http://www.boost.org`, 2009.

[4] CWSandbox. `http://www.cwsandbox.org/`, 2009.

[5] Description of Allaple by F-Secure. `http://www.f-secure.com/v-descs/allaple_a.shtml`, 2009.

[6] Description of Trojan-PWS.Win32.LdPinch by Mcafee. `http://vil.nai.com/vil/content/v_100539.htm`, 2009.

[7] Description of Virus.Win32.Virut by F-Secure. `http://www.f-secure.com/v-descs/virus_w32_virut.shtml`, 2009.

[8] JoeBox. `http://www.joebox.org`, 2009.

[9] Norman Sandbox. `http://www.norman.com/microsites/nsic/`, 2009.

[10] ThreatExpert. `http://www.threatexpert.com/`, 2009.

[11] 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.

[12] 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.

[13] U. Bayer, P. Milani Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[14] 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.

[15] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Usenix Security Symposium*, 2003.

[16] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.

[17] 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.

[18] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.

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

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

[21] P. Jaccard. The Distribution of Flora in the Alpine Zone. *The New Phytologist*, 11(2):37–50, 1912.

[22] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Application Conference (ACSAC)*, 2004.

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

[24] 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.

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

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