MalGene: Automatic Extraction of Malware Analysis Evasion Signature

Dhilung Kirat
University of California, Santa Barbara
dhilung@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Automated dynamic malware analysis is a common approach for detecting malicious software. However, many malware samples identify the presence of the analysis environment and evade detection by not performing any malicious activity. Recently, an approach to the automated detection of such evasive malware was proposed. In this approach, a malware sample is analyzed in multiple analysis environments, including a bare-metal environment, and its various behaviors are compared. Malware whose behavior deviates substantially is identified as evasive malware. However, a malware analyst still needs to re-analyze the identified evasive sample to understand the technique used for evasion. Different tools are available to help malware analysts in this process. However, these tools in practice require considerable manual input along with auxiliary information. This manual process is resource-intensive and not scalable.

In this paper, we present MalGene, an automated technique for extracting analysis evasion signatures. MalGene leverages algorithms borrowed from bioinformatics to automatically locate evasive behavior in system call sequences. Data flow analysis and data mining techniques are used to identify call events and data comparison events used to perform the evasion. These events are used to construct a succinct evasion signature, which can be used by an analyst to quickly understand evasions. Finally, evasive malware samples are clustered based on their underlying evasive techniques. We evaluated our techniques on 2810 evasive samples. We were able to automatically extract their analysis evasion signatures and group them into 78 similar evasion techniques.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Software Engineering**]: Security and Protection—*Invasive software (malware)*; J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813642 .

Keywords

computer security; malware analysis; evasive malware; sequence alignment; bioinformatics

1. INTRODUCTION

Automated dynamic malware analysis is a common approach for analyzing and detecting a wide variety of malicious software. Dynamic analysis systems have become more popular because signature-based and static-analysis-based detection approaches are easily evaded using widely available techniques such as obfuscation, polymorphism, and encryption. However, many malware samples identify the presence of the analysis environment and evade detection by avoiding the execution of suspicious operations. Malware authors have developed several ways to detect the presence of malware analysis systems [13, 25, 26, 28, 29]. The most common approach is based on the inspection of some specific artifacts related to the analysis systems. This includes checking for the presence of registry keys or I/O ports, background processes, function hooks, or IP addresses that are specific to some known malware analysis service. For example, a malware running inside a Virtualbox *guest* operating system can simply inspect Virtualbox-specific service names, or the hardware IDs of the available virtual devices, and check for the substring `VBOX`. Another approach to evasion is to fingerprint the underlying CPU that is executing the malware. For example, fingerprinting can be achieved by detecting the differences in the timing property of the execution of certain instructions, or a small variation in the CPU execution semantics [25, 29].

Recently, an approach to the automated detection of evasive malware has been proposed [17]. In this approach, malware is executed in a bare-metal execution environment as well as environments that leverage virtualization and emulation. Malware behaviors are extracted from these executions and compared to detect deviations in the behavior in the assumption that bare-metal execution represents the "real" behavior of the malware. Malware whose behavior deviates substantially among the execution environments is labeled as evasive malware. This way, evasive malware is identified without knowing the underlying evasion technique. This approach requires each malware to be run on a bare-metal environment. However, compared to a bare-metal environment, emulated and virtualized environments are easier to scale and they provide far better control and visibility over malware execution. For these practical reasons, emulated or virtualized sandboxes are widely used for large-scale automated malware analysis. However, keeping

up emulated and virtualized sandboxes resistant to evolving evasion techniques is a current industry challenge. To combat sandbox evasion attacks, a complete understanding of evasion techniques is the first fundamental step, as this knowledge can help "fix" sandboxes and make them robust against evasion attacks. Currently, understanding evasion techniques is largely a manual process.

Several analysis tools are available to help analyze malware behavior differences [7, 15]. These tools are effective in performing manual, fine-grained analysis of evasive malware. However, they require additional auxiliary information, such as a set of system calls corresponding to malicious behavior or the selection of control-flow differences. Finding this auxiliary information is a manual process. This manual process is resource-intensive and not scalable. However, performing such analysis on a large scale is necessary to combat rapidly evolving evasion attacks.

In general, the manual process required to understand an evasion instance starts from two sequences of system call traces of the same malware sample when executed in two different execution environments. The malware sample evades one of the environments, creating a difference between the system call sequences. The first step of the evasion analysis involves finding the location in the system call traces where the execution deviates due to evasion. After accurately locating the deviation, understanding the evasion requires identifying environment-specific artifacts that are used for fingerprinting the analysis environment. In the first step, manually finding the location of the deviation in the system call sequence can be difficult. The naïve approach of looking for the first call that is different in both sequences does not work. System call traces are usually noisy, and there can be thousands of events in the sequence. Even when running the same program in exactly the same environment twice, the system call traces can be quite different. Thread scheduling is one of the main reasons for these differences, however, other factors, such as operating-system and library-specific aberrations, initialization characteristics, and timing, can play a substantial role. Another approach would be to take a *diff* of the sequences, in the assumption that there will be a large gap in the alignment corresponding to the evasion in one of the environments. However, this approach may not accurately align the sequences. A generic *diff* algorithm finds the longest common subsequence (LCS) of the sequences. This approach is effective when large portions of the sequences have unique *alphabets*, such as the lines of a source code. However, a system call sequence has a limited alphabet, while the sequence itself is usually long. Because of this, instead of forming a gap, some subsequence of system calls corresponding to malicious behaviors are likely to align with another sequence where the malicious behavior is absent.

In this paper, we present MalGene, an automatic technique for extracting human-readable evasion signatures from evasive malware. MalGene leverages local sequence alignment techniques borrowed from bioinformatics to automatically locate evasions in a system call sequence. Such sequence alignment techniques are widely used for aligning long sequences of DNA or proteins [11, 14, 27]. These algorithms are known to be effective even if there are large gaps and the size of the alphabet is limited, such as the alphabet of four bases: Thymine (T), Adenine (A), Cytosine (C), and Guanine (G) in case of DNA sequence. We use data flow analysis and inverse document frequency-based techniques to automatically identify call events and data comparisons used by the evasion techniques. We build evasion signatures from these identified events. Finally, malware samples are clustered based on their underlying evasive techniques.

Our work makes the following contributions:

- We present MalGene, a system for automatically extracting evasion signatures from evasive malware. Our system leverages a combination of data mining and data flow analysis techniques to automate the signature extraction process, which can be applied to a large-scale sample set.
- We propose a novel bioinformatics-inspired approach to system call sequence alignment for locating evasions. The proposed algorithm performs deduplication, difference pruning, and can handle branched sequences.
- We evaluated our techniques on 2810 evasive samples. We were able to automatically extract their analysis evasion signatures and group them into 78 similar evasion techniques.

2. EVASION SIGNATURE MODEL

In general, malware evades analysis in two steps. First, it extracts information about the execution environment. Second, it performs some comparison on the extracted information to make the decision whether to evade or not. Usually, malware uses system calls and user-mode API calls in the first step to probe the execution environment. In the second step, it uses some predefined constant values or information extracted from previous system or user API calls. With this generalization, we define an *evasion signature* as a set of system call events, user API call events, and comparison events that are used as the basis for evading the analysis system. A comparison event is an execution of a comparison instruction, such as a CMP instruction in the x86 instruction set. Usually, a call to one of such instructions is necessary to make the control flow decision during evasion, which is the second step of the evasion process as mentioned earlier.

Formally, let P be the set of all call events (both system calls and API calls) and Q be the set of all comparison events that are used by an evasion technique; we define the evasion signature Δ of this technique as:

$$\Delta = P \cup Q$$

We represent a call event $p : p \in P$ as a pair $(name(p), attrib(p))$, where, $name(p)$ represents the name of the call, e.g., `NtCreateFile`, and $attrib(p)$ represents the name of the operating system object associated with the call, e.g., `C:/boot.ini`. We represent a comparison event $q : q \in Q$ as a pair (p, v), where p is a call event that produced the information in the first operand compared by event q. v represents either some constant value used in the second operand, or another call event that produced the information for the second operand.

We extract the evasion signature Δ of an evasive malware sample in two steps. In the first step, we locate the evasion in the call sequences resulting from the execution of the malware in different environments, as described in Section 3.2.3. In the second step, we identify the elements of Δ used for the evasion, as described in Section 4.

The model defined by Δ only captures those evasion techniques that must trigger some system or user API calls. The majority of known evasion techniques falls in this category. Some techniques may not directly make a system or API call, such as a forced exception-based CPU-fingerprinting [25]. However, such techniques indirectly trigger calls to exception handlers, which are captured by P. But again, in case of an emulated CPU, there are known evasion techniques that are entirely based on the inspection of the FPU, memory, or register state after the execution of certain instructions. Some evasion techniques are based on stalling code. Our current model does not capture such evasion techniques.

3. SEQUENCE ALIGNMENT

The input to our system is a set of evasive malware samples detected by an automatic evasion detection systems, called BareCloud [17]. BareCloud provides information about which of the analysis environments a malware sample evades. To extract the evasion signature of an evasive malware sample, we analyze the sample in two analysis environments where it evades one of the environments while showing malicious activity in the other. In the first step, we start from the two sequences of system call events from these two analysis environments. Because the system calls related to the malicious activities are entirely missing in one of the sequences, there must be an observable deviation between the two sequences. The goal here is to efficiently and accurately find the location of the deviation in the sequence corresponding to the evasion. To do this, we first align two sequences starting from the beginning, introducing gaps as required for an optimal alignment. We locate the deviation by finding the largest gap in the aligned sequence. We consider this location as the *evasion point*. The malware activity significantly differs after this point, implying evasion.

The intuition here is that an evasive malware sample must perform its evasion "check" in both environments before the *evasion point*. Once we locate the *evasion point*, we extract the *evasion signature* from the detailed analysis log, which contains user API calls and comparison events, as described in Section 4. Note that only the system-call level monitoring is required for locating the *evasion point*. This is advantageous because the monitoring of user API calls and comparison events may not be available in both analysis environments. However, most of the existing malware analysis systems are capable of producing system-call level execution profiles.

Apart from the malware evasion, there can be other factors that can cause deviation in the malware execution. We followed all strategies proposed in BareCloud [17] to limit deviations due to external factors. That is, we used identical local network and identical internal software configurations for all execution environments. We executed each malware sample in both environments at the same time to mitigate date time-related deviations. We used network service filters to provide consistent responses to DNS and SMTP communications for all environments.

One simple approach to finding the largest gap in the alignment of system call sequences would be to take a *diff* of the sequences. However, the generic *diff* algorithm finds the longest common subsequence (LCS) of the sequences, which may not accurately align the sequences in our context. This is because a) a system call sequence is usually a long series of events drawn from a limited alphabet, e.g., around 300 system calls in the Windows platform, and b) the difference between the sequences tends to be large. System call names when combined with their arguments can increase the size of the alphabet. However, there are frequent system calls, such as `NtAllocateVirtualMemory`, that act on unnamed OS objects or nondeterministic argument values. Using nondeterministic argument values, such as memory addresses, creates too many undesirable mismatches resulting a poor alignment. In such cases, we discard the *attrib*() values of the system call events to get a more stable alignments. To illustrate this, let us take example sequences A and B as shown in Figure 1(a). Here, sequences A and B are system call sequences of the same malware sample when executed in two different execution environments. Sequence A corresponds to the execution environment where the malware evades analysis, while sequence B corresponds to the execution environment where the malware shows its malicious activity. The "malicious section" of the sequence B corresponding to the malicious activity of the malware sample is illustrated with a darker background. This malicious section is missing in the sequence A because the malware sample evades analysis. In this example, the LCS-based alignment matches the first three calls from $A1$ with $B1$, as expected. However, the rest of the sequence of A is matched with common subsequences from the malicious section of B to maximize the length of the common subsequence. In this case, it is an algorithmically optimal but semantically incorrect alignment. However, this is likely to happen because the malicious sections are usually long and the alphabet is limited in size. Note that the system call `NtTerminateProcess` does not align because such alignment will result in a shorter common subsequence. However, the alignment of important call events is critical for accurately locating the *evasion point*. This LCS-based alignment example shows that the longest common subsequence may not always produce the most meaningful alignment of the system call sequences.

To address this problem, we propose to apply sequence alignment algorithms borrowed from bioinformatics. Such algorithms are used to identify regions of similarity in sequences of DNA, RNA, or proteins [11, 14, 27]. These regions of similarity usually correspond to evolutionary relationships between the sequences [22]. In the case of system call sequences, such similarity regions correspond to the execution of similar code or the same high-level library functions. While aligning system call sequences, the alignments of some system calls are more critical than others, such as the alignment of `NtTerminateProcess` in Figure 1(a), because they represent important events in the program execution. Sequence alignment algorithms from bioninformatics can prioritize such critical alignments. Furthermore, these algorithms support more versatile similarity scores among system calls, which can produce better approximation of the alignments in the presence of noise in the sequences.

There are two approaches to sequence alignment: Global Alignment and Local Alignment. In the next section, we briefly describe these approaches.

3.1 Global and Local Alignments

When finding alignments, global alignment algorithms, such as Needleman-Wunsh [24], take the entirety of both sequences into consideration. It is a form of global optimization that forces the alignment to span the entire length [27].

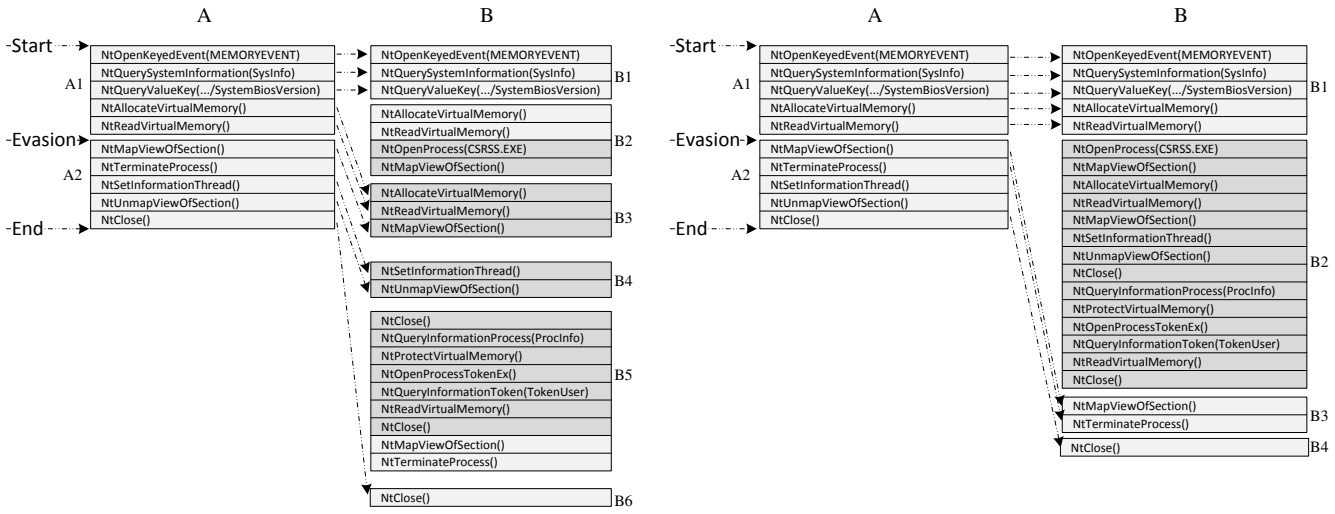

Figure 1: Sequence Alignments

This approach is useful when there is no deviation in the malware behavior, or the deviation is minimal.

Local alignment algorithms, such as Smith-Waterman [30], tend to find good matches of local subsequences between two sequences. Hence, these algorithms identify regions of similarity within long sequences that are often widely divergent overall. This approach is better if there are large missing parts in the sequence. This is true for a system call sequence corresponding to evasion, such as sequence A in Figure 1(b), which is missing system calls corresponding to $B2$, the malicious section of B. For this reason, we use a Local Alignment algorithm for aligning system call sequences. Figure 1(b) represents the alignment using a local alignment algorithm. Notice that there is no undesirable alignment with the malicious section of the sequence B. The `NtTerminateProcess` system call is aligned even though the total number of matches is smaller compared to the LCS-based alignment (8 vs. 9 matches). The alignment in Figure 1(b) is clearly the better alignment for locating the *evasion point* compared to the LCS-based alignment in Figure 1(a).

3.1.1 Local Alignment

In this section, we briefly describe the Smith-Waterman [30] local alignment algorithm.

Given two sequences $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$ of length n and m respectively, a *maximum similarity* matrix H is computed using the following induction:

$H(i, 0) = 0,\ 0 \leq i \leq m$,

$H(0, j) = 0,\ 0 \leq j \leq n$,

and

$$H(i,j) = \max \left\{ \begin{array}{c} 0 \\ H(i-1, j-1) + \ Sim(a_i, b_j) \\ \max_{k \geq 1}\{H(i-k, j) + \ W_k\} \\ \max_{l \geq 1}\{H(i, j-l) + \ W_l\} \end{array} \right\},$$

$1 \leq i \leq m, 1 \leq j \leq n$

where a and b are strings over the alphabet Σ, $Sim(a, b)$ is a similarity score function on the alphabet, and W_i is the *gap penalty* schema. Here, $H(i, j)$ represents the maximum similarity score between suffixes of $[a_1, a_2...a_i]$ and $[b_1, b_2...b_i]$. To obtain the optimal local alignment, backtracking is performed starting from the highest value in the matrix $H(i, j)$.

We used a scalable implementation of the local alignment algorithm [14]. We provide more information about the similarity score function and gap penalty schema in the next sections.

3.2 System Call Alignment

A system call sequence consists of a sequence of system call events. While the order of biological sequences represents a structural property, the order of system call sequence represents the temporal execution order. The order of system call events has stronger significance when events are interdependent. For example, in order to create a thread in a foreign process to run arbitrary code, one must follow a certain order of system calls. Even with insertion of gaps, sequence alignment preserves this order while aligning sequences.

3.2.1 Similarity Score

One of the most important parts of the sequence alignment algorithm is the similarity-scoring schema. Based on the domain knowledge, the scoring schema computes a similarity score between two elements in the sequence. A straightforward approach would be to simply assign a value $\mu > 0$ for a *match* and $\sigma < 0$ for a *mismatch*. Values of μ and σ can be constant values or they may depend on the pair of sequence elements being compared.

There are many studies on modeling similarity schema for biological sequence alignment [3, 9]. These schemata are based on biological evidence, where a *mismatch* is treated as *mutation*. In general, the match score μ is based on the functional significance of the match, and the mismatch score σ is statistically computed from the observed mutations seen in nature. Point Accepted Mutation (PAM) [9] and Blocks Substitution Matrix (BLOSUM) [3] are the two most widely-used similarity schemata. The main focus of these schemata is to model mismatch scores based on the observed probability of the mutation under comparison. A similar approach may be useful while comparing system call

sequences of polymorphic variants of malware. However, we are comparing system call sequences of the same code. We observed that malware polymorphism happens mostly during the propagation step, i.e., while the malware sample creates a copy of itself, while runtime polymorphism is less common. Moreover, achieving the same functionality by replacing the system call is difficult. That is, the probability of mutation in the system call sequences extracted from two executions of the same malware sample is very small. This means that the mismatches of system calls are less common.

In our case, the challenge is to meaningfully quantify *match* and *mismatch* in case of system calls. There may be a varying number of *arguments* associated with each system call event. Not all arguments are equally important for similarity computation. As discussed earlier, alignments of some system calls are more important than others. For example, we want to prioritize the alignment of `NtCreateProcess` over `NtQueryValueKey` because creating a process is a more critical event compared to reading a registry value. We can assign a high similarity value for a match of a critical system call, which helps build an "anchor point" during the alignment process. In our current model, the list of such critical system calls includes system calls that create and terminate processes and threads. We propose the following similarity-scoring schema for computing similarity between two system calls.

$$Sim(a,b) = Bias(a,b) * (NameSim(a,b) + AttribSim(a,b))$$

where,

$$NameSim(a,b) = \left\{ \begin{array}{ll} w_t & \text{if } name(a) = name(b), \\ nw_t & \text{if } name(a) \neq name(b) \end{array} \right\}$$

,

$$AttribSim(a,b) = \left\{ \begin{array}{ll} w_a & \text{if } name(a) = name(b) \\ & \text{and } attrib(a) = attrib(b), \\ nw_a & \text{if } name(a) = name(b) \\ & \text{and } attrib(a) \neq attrib(b), \\ 0 & \text{If } name(a) \neq name(b) \end{array} \right\}$$

, and

$$Bias(a,b) = \left\{ \begin{array}{ll} w_b & \text{if } name(a) \text{ or } name(b) \\ & \text{is an important system call,} \\ 1 & \text{else.} \end{array} \right\}$$

Here, a and b are system call events, and $name()$ and $attrib()$ have the meaning described in Section 2. In practice, similar system calls are those calls that perform similar actions on similar operating system objects.

3.2.2 Gap Penalty

Another important component of the sequence alignment algorithm is the gap penalty schema. In general, a gap penalty is a negative score added to the similarity score to discourage *indels* (insertion or deletion). Large gap penalty is effective in aligning sequences properly if the majority of the sequences are identical. However, in our case, we expect to have gaps in the sequence because of the noise and the evasion. Since our goal is to properly identify the gap introduced by evasion, in some way we want to encourage long gaps in the alignment.

There are three main types of gap penalties used in the context of biological sequences: constant, linear, and affine gap penalty. The constant gap penalty simply gives a fixed negative score for each gap opening. This value does not depend on the length of the gap. This is a simple and fast schema. However, this schema gives too much freedom for sequence alignment, resulting in unnecessary long gaps. The linear gap penalty, as the name implies, linearly increases the penalty score in proportion to the length of the indel. This method favors shorter gaps by severely penalizing long indels, which is not suitable in our context. The affine gap penalty combines both constant and linear gap penalties, taking the form $g_a + (g_b * L)$. That is, it assigns an opening gap penalty g_a, which increases with the rate of g_b. We can use a smaller value for g_b to favor longer gaps. By choosing $|g_a| > |g_b|$, we can model a gap penalty such that it is easier to extend a gap than to open it. We use this model of gap penalty when aligning system call sequences.

3.2.3 Parameter Selection

In our approach to system call alignment, like any other alignment problem, there are certain constraints we need to follow while designing similarity score and gap penalty parameters. More precisely, we want to have the following inequality relation as a guideline for choosing parameter values:

$$nw_t \leq g_a < g_b < 0 < w_t + nw_a < w_t < w_t + w_a$$

Here, we want all mismatches and indels to have negative values and all matches, including partial matches, to have positive values. Intuitively, a match where both $name()$ and $attrib()$ match gets the highest score $(w_t + w_a)$. Similarly, a $name()$ match and $attrib()$ mismatch gets a lower score $(w_t + nw_a)$ than when $name()$ matches but there is no $attrib()$ associated with the events to be compared with, such as for the `NtYieldExecution` system call event. By choosing $nw_t \leq g_a$, we favor gaps over mismatched alignment. The inequality relation among parameters and their relative values are more important than the actual values of the parameters. If all parameters are scaled by the same factor, the final alignment output of the algorithm remains the same.

Furthermore, the bias multiplier w_b used to compute $Bias(a,b)$ needs to be large enough to overcome possible penalty introduced by expected long gaps in case of evasive samples. For example, we want to prioritize the alignment of the `NtTerminateThread` system call, which is usually located towards the end of the sequences, which requires a long gap.

3.2.4 Deduplication

Sometimes a tight loop in the execution may produce a long sequence of repeated short subsequences. Such repetition may contain thousands of system calls, excessively increasing the space and time complexity requirement for sequence alignment. To this end, we identify contiguously repeating subsequence of system calls of length one, two, and three. If such subsequence repeats more than five times contiguously, we discard all remaining subsequences during sequence alignment. There are two advantages in doing this. First, it greatly reduces the space and time requirement for sequence alignment. Second, it prevents possible inaccurate

detection of the longest gap due to the difference in the repetition count between two call sequences.

3.2.5 Difference Pruning

Accurately identifying the largest gap corresponding to the evasion is critical in finding the evasion point. However, there is a possibility of short subsequence alignments breaking the large gap associated with the evasion. This may cause the algorithm to incorrectly pick the largest gap and, in turn, return the incorrect evasion point. To mitigate this problem, we apply a difference pruning process to prune possibly-incorrect small alignments in-between large gaps. Let S be the sequence alignment output of two sequences A and B. Let S_a, S_b, and S_c be three consecutive regions of S where, S_b is a *match* alignment and S_a and S_c are *gap* alignments such that they are both insertion or both deletion alignments. We discard the S_b alignment region to combine two gaps corresponding to S_a and S_c if S_b is relatively very small compared to the length of S_a and S_c combined. More precisely, if $length(S_b)/(length(S_a)+length(S_c)) < t_d$ we discard S_b and join S_a and S_c to find the largest gap. Through a series of experiments described in Section 5, we obtained the optimal value of $t_d = 0.02$. That is, if the length of a match region between two gaps is less than 2% of the sum of the lengths of the gaps, we prune the match region and connect the gaps. This process only affects the calculation of the largest gap without affecting the actual sequence alignment output. The pruning is performed in a single pass without updating the underlying sequence to avoid a newly-formed longer gap destabilizing the pruning process.

3.3 Handling Sequence Branching

All sequence alignment algorithms from bioinformatics can only handle monolithic single sequences. However, a sequence of system calls may include calls from multiple threads. The main process thread can create multiple threads, which in turn can create more threads. Hence, a system call sequence has an inherent tree structure. A naïve way of combining system calls from multiple threads into a single call sequence can produce anomalous sequence alignment.

We propose a recursive algorithm to handle branched sequences. The input of this algorithm is a single system call sequence of a process where system calls from all threads are chronologically merged. Each event in the sequence is tagged with its corresponding thread ID. First, we preprocess this system call sequence to generate a branching sequence structure by sequentially inspecting events from the start of the sequence. Whenever a new thread is encountered, we insert a new *meta-node* at the location where the thread was created. This is the location of the `NtCreateThread` system call corresponding to the thread. We create a new blank sequence and associate it with the new meta-node. A meta-node represents a branching point in the main sequence. We remove all occurrences of system calls associated with the new thread from the main sequence and append it to the newly created sequence associated with the *meta-node*. The one-to-one mapping of a new thread event and its corresponding `NtCreateThread` may not always be available in the execution profile. To this end, we assign a new thread event with the last unassigned call of `NtCreateThread`. During the alignment process, two meta-nodes are recursively processed to compute the similarity score. That is, to compute the similarity score between two meta-nodes, we first perform sequence alignment of the sequences corresponding to the meta-nodes. Similarity is then computed as the difference between the total length of the matching sections and the total length of the mismatch sections of the alignment output. If at least one of the two arguments to $Sim(a, b)$ is a meta-node, the following similarity-scoring schema is used.

$$Sim(a, b) = MSim(a, b) \tag{1}$$

where,

$$MSim(a, b) = \left\{ \begin{array}{ll} s_m - s_g & \text{if } a \text{ and } b \text{ are meta-nodes,} \\ -n_a & \text{if only } a \text{ is a meta-node,} \\ -n_b & \text{if only } b \text{ is a meta-node.} \end{array} \right\}$$

.

Here, s_m is the total length of all matching sections of the alignment output corresponding to meta-nodes a and b, s_g is the total length of all gap sections of the alignment output, n_a is the length of the sequence corresponding to meta-node a, and n_b is the length of the sequence corresponding to meta-node b. Note that if meta-nodes a and b correspond to two completely different threads, s_g will be greater than s_m in resulting a negative similarity score.

4. EVASION SIGNATURE EXTRACTION

In the previous section, we described an *evasion point* as the location of deviation in the call event sequence corresponding to the evasion. In this section, we describe how we use this information to extract the evasion signature.

4.1 Evasion Section

Intuitively, all system calls, API calls, and comparison events used to make an evasion decision must happen before the evasion point. We observed that such events are usually located close to the evasion point. To capture the locality of such events in the sequence, we define an *evasion section*, which consists of the event sequence prior to but close to the evasion point. More precisely, let E be the sequence of malware execution events that consists of all system call events, user API call events, and comparison events, the evasion section $E^{'}$ of the event sequence E is defined as:

$$E^{'} = \{e \in E(i) : k - \omega \leq i < k\},$$

where, k is the index to the evasion point, and ω is the size of the evasion section.

If ω is large enough, $E^{'}$ can extend all the way to the beginning of the event sequence E. This case guarantees that $P \subset E^{'}$ and $Q \subset E^{'}$ where, P and Q are call events and comparison events related to evasion, as introduced in Section 2. That is, evasion signature $\Delta \subset E^{'}$, since $\Delta = P \cup Q$. However, with large values of ω, evasion section $E^{'}$ also includes many other events that are not related to evasion. By reducing the value of ω we can reduce the number of such unrelated events and improve the relation $\Delta \approx E^{'}$. We also observed that the comparison events in Q that are used for evasion are likely to be performed very close to the evasion point k. This allows us to reduce ω to smaller values and still have $Q \subset E^{'}$. This approach might exclude call events made earlier in the sequence whose results are used later for evasion. To mitigate this, we include all call events that are related to comparison events in Q into the evasion signature.

Notice that, unlike the previous sequence alignment step, in this step a call event includes both system calls and API calls. Although many user API calls correspond to system calls, many user mode APIs may not trigger any system call. For example, the user mode API `GetTickCount` in Windows does not invoke any system call (native API). However, this API is widely used in timing-based evasions. We must include such call events in the evasion signature to make it more accurate and complete.

Initially, we set P as the set of all call events in the evasion section $E^{'}$, and Q as the set of all comparison events in $E^{'}$. However, even with smaller values of ω, the evasion section $E^{'}$ still contains unrelated call events. In the next section, we describe our approach to filtering out these unrelated events using statistical observations.

4.2 Inverse Document Frequency

A call event used to retrieve information from the analysis environment for fingerprinting is usually unique to the evasive behavior. The majority of the malware samples that are not evasive do not retrieve those unique pieces of information. Similarly, if the same call event e (same $name(e)$ and $attrib(e)$) is present in the call sequences of all non-evasive malware, such call event is less likely to be used for evasion. We can filter out call events from the evasion section $E^{'}$ that occur too often in the collection of call sequences of non-evasive malware. To perform such filtering, we use inverse document frequency-based metric.

Inverse document frequency (idf) is commonly used in information retrieval [31]. It is a measure of whether a term is common or rare across all documents. Formally, the inverse document frequency of a term t in a collection of documents D is defined as:

$$idf(t, D) = \log \frac{N}{df_t}$$

where, N is the total number of documents in the corpus and df_t is the document frequency, defined as the number of documents in the collection D that contain the term t.

In our case, a call event is a term, and collection of call sequences of non-evasive malware is the document corpus D. For a call event e, a large value of $idf(e, D)$ implies that the call event e is unique, and a small value of $idf(e, D)$ implies that e is commonplace. Here, $idf(e, D) = 0$ means the call event e is present in all call sequences of D.

We define a threshold τ such that, if $idf(e, D) < \tau$, we consider the call event e to be a common event having little or no discriminating power for building evasion signatures. We remove such call events $\{e : idf(e, D) < \tau\}$ from P.

4.3 Event Dependency Analysis

The next component of the evasion signature is the comparison events Q used for altering control flow during evasion. Comparison events can be monitored with any fine-grained instruction-level execution monitoring. However, we are interested in only those comparisons that involve the use of information generated by previous call events. To track the information returned by call events we leverage taint analysis. To this end, we build upon the work of the Anubis extension proposed in [5]. Anubis [1] is a malware analysis framework, which uses Qemu-based full-system emulation as the execution environment. In this approach, information returned by all call events is tainted at the byte level. Inside Qemu intermediate language, all comparison instructions of x86 architecture are translated into the same intermediate comparison instruction. For each comparison, taint labels of the operands are examined to determine corresponding call events that produced the data byte. Consecutive comparisons are merged into a single comparison event. In case the comparison in performed with some constant, the constant value is also extracted.

Beside taint analysis, we also analyze *handle* dependencies between call events. This allows us to generate a more descriptive value of $attrib(e)$ for the call event e. For example, if a registry key `HKLM/System` is opened by a call to `NtOpenKey` and the returned handle is later used for a call to `NtEnumerateKey`, we use the registry key name as the $attrib(e)$ for the call event `NtEnumerateKey`.

An execution of a program contains many comparisons even if only comparisons with tainted operands are considered. However, many of such comparisons originate from within API functions rather than the actual malware code. For this reason, comparisons inside user API calls are discarded, except for API calls that are designed specifically for data type comparison, such as strings and dates. Comparison events are included in the execution profile of the malware along with the system call and user API call events.

We build the sequence of malware execution events E from the execution profile generated by Anubis. We also extract the system call sequence from another execution environment that the malware evaded. Since the evasion code executes in both environments, we can extract its evasion signature from Anubis execution profile regardless of which environment is evaded. We identify the evasion point and evasion section $E^{'}$ using the approach described in the previous sections. We extract the call events P and the comparison events Q from the evasion section $E^{'}$. We filter P using the idf-based method described previously.

Finally, all call events associated with Q are added to the set P. The union of P and Q represents our final evasion signature $\Delta = P \cup Q$.

4.4 Clustering

Given a collection of evasive samples, we propose to assess different evasion techniques present in the collection based on the extracted evasion signatures. To do this, we perform hierarchical clustering of evasive samples. This allows a malware analyst to prioritize and selectively study different evasion techniques without analyzing randomly selected samples. To perform manual assessment of a particular cluster, we can take an intersection of the evasion signatures of all samples from that cluster. That is, we inspect the evasion signature elements that are common to all samples in the cluster.

A hierarchical clustering requires a method to compute pairwise similarity between two evasion signatures. An evasion signature is essentially a set. We compute similarity between two evasion signatures Δ_a and Δ_b as a Jaccard Similarity J, which is given as:

$$J(\Delta_a, \Delta_b) = \frac{\mid \Delta_a \cap \Delta_b \mid}{\mid \Delta_a \cup \Delta_b \mid}.$$

The result of a hierarchical clustering depends on the choice of the linkage method and the similarity measure, where, the former is usually more critical than the latter [32]. There are two main choices of linkage methods; single-linkage

and complete-linkage. We use the complete-linkage method for our clustering. This is because the complete-linkage method prefers compact clusters with small diameters over long, straggly clusters [21]. As we want maximum similarity between all pairs of members in a cluster for assessment, the complete-linkage method best fits our purpose.

5. EVALUATION

We evaluated our approach on real-world Windows-based evasive malware samples. We made this choice because the majority of the evasive malware is observed on this platform. Moreover, the majority of the malware analysis systems are also focused on the same platform.

5.1 Execution Environments

In our evaluation, we provide two execution environments based on emulation and hardware virtualization, respectively.

5.1.1 Emulation

We use Anubis [1] to extract malware execution events from an emulated environment. Anubis performs execution monitoring by observing an execution of precomputed guest memory addresses. These memory addresses correspond to system call functions and user API functions. Anubis is able to extract additional information about the API execution by inserting its own instructions to the emulator's instruction execution chain. Besides system calls, we are able to extract additional information, such as user API calls and comparison events, which are necessary for building evasion signatures.

5.1.2 Hypervisor

We use Ether [10] to extract malware execution events from a hardware-based virtualized environment. Ether is a Xen-hypervisor-based transparent malware analysis framework that utilizes Intel VT's hardware virtualization extensions [2]. The hardware virtualization makes it possible to execute most of the malware instructions as native CPU instructions on the real hardware without any interception. Thus, it does not suffer from inaccurate or incomplete system emulation. It was observed that Ether can be evaded in its default setup because it uses QEMU's device model to provide virtualized hardware peripherals [17]. We modified the device model used by Ether to prevent such evasion.

5.2 Dataset

The input for our system is a collection of known evasive malware samples. For the evaluation of our system, we received 3,107 evasive samples identified by the BareCloud [17] system. We analyzed those samples in Anubis and our modified Ether environments. We extracted system call traces and computed behavior deviation scores as proposed in [17]. We found that 2810 samples evaded Anubis with respect to our Ether environment.

To build the ground truth dataset, we randomly selected 52 samples out of 2810 evasive samples. We manually analyzed those samples and identified the calls and the comparisons that are related to the evasion. This information constitutes the evasion signature Δ of the malware samples. To evaluate the alignment algorithm, which works only on the system call sequences, we identified the most *important* system call that is critical to the evasion technique as the *evasion call*. In case multiple related system calls are used, we selected the last system call as the *evasion call*. For instance, let us take an example evasion instance that opens a registry key `HKLM/HARDWARE/Description/System` using `NtOpenKey` and reads the value of the key `SystemBiosVersion` using `NtQueryValueKey`. Inside Anubis, the returned value is `QEMU -1` because of the underlying Qemu subsystem, which can be checked for evasion. In this example, both system calls are related to evasion. However, we select the last call to `NtQueryValueKey` as the *evasion call*. We note the index of this instance of the system call in the sequence as the data point used later in the experiments.

5.3 Algorithm Evaluation

In our approach, accurately finding the *evasion point* is the first and critical step towards extracting evasion signatures. This depends on the accuracy of the proposed sequence alignment algorithm for system calls. The accuracy depends on several parameters used by the algorithm. In Section 3.2, we discussed some guidelines for choosing optimal parameters for algorithm. However, there is no previous work on this area. Unlike in the field of bioinformatics, an appropriate labeled dataset is lacking to build a statistical model of similarity score for system call sequences. We use an incremental approximation-based approach to find optimal values of the parameters, which we describe in the next section.

5.3.1 Experiment with Scoring Function

To evaluate our guideline, we performed several experiments by varying different scoring parameters. For this, we first chose to vary a set of four main parameters (g_a, g_b, nw_t, w_t, see Section 3.2). Our preliminary experiments showed that the values of these parameters play a major role in the algorithm output. For the remaining parameters, we empirically assigned constant values. For each set of parameter values (g_a, g_b, nw_t, w_t) we performed sequence alignment to find the corresponding *evasion point*. Let A_m be a sequence corresponding to a malware m and let k_m be the index to the calculated *evasion point*. Let, $e_m^{'}$ be the index to the *evasion call* in A_m, which is known as the ground truth. We say that an *evasion section* of width $w^{'}$ successfully captures the *evasion call* if $k_m - w^{'} \leq e_m^{'} < k_m$. That is, the *evasion call* is within the *evasion section* defined by $w^{'}$. For a set of N samples, we compute the recall rate corresponding to the parameter set (g_a, g_b, nw_t, w_t) and *evasion section* $w^{'}$ as TP/N, where TP is the number of samples that are within the *evasion section* defined by $w^{'}$.

Figure 2 shows the results of the recall rate of some parameter sets when varying the *evasion section* of width $w^{'}$. We used the ground truth dataset as described in Section 5.2. The area under the curve (AUC) represents the relative performance of the choice of the parameters. The result validates some of our initial intuitions. For example, choice of $\mid g_a \mid > \mid nw_t \mid$ decrease the algorithm performance (compare top and second curves), a relatively large score for a match compared to the gap penalty degrades performance (third curve), and a large gap extension penalty g_b is not favorable (top and bottom curves).

There are many possible combinations of parameter choices. To find the optimal choice, we computed AUC values for all possible combinations when g_a, g_b, nw_t, and w_t are selected from the sets 10 values for each parameter ranging from -10

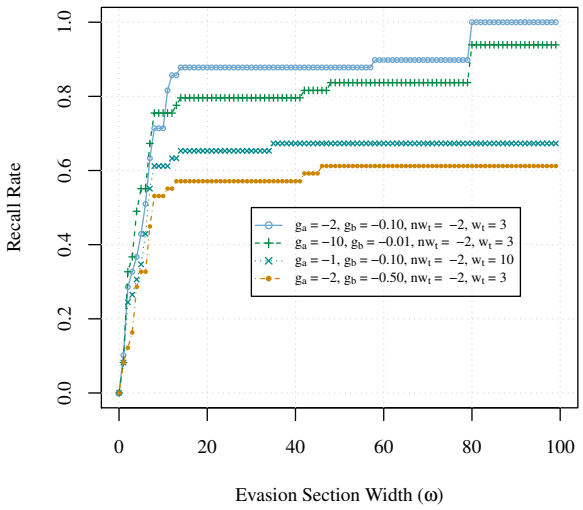

Figure 2: Parameter tuning experiment. (w_a = 2 and nw_a = -2).

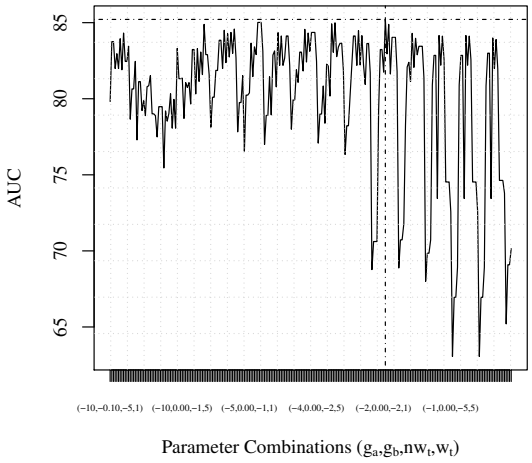

Figure 3: Average AUC values of recall rates corresponding to 10,000 parameter combinations.

to +10. That is, for each malware sample there are 10,000 test cases. To test the correctness of our guidelines, we also included values that do not satisfy the inequality guideline. Figure 3 shows the result of the AUC values for all parameter combinations. The combination of parameters $g_a = -2$, $g_b = -0.10$, $nw_t = -2$, $w_t = 3$ produces the highest value of AUC (=85.214).

In the next step, we performed another set of similar experiments by varying the values of other parameters while keeping the values of g_a, g_b, nw_t, and w_t set to the optimal values obtained from the previous experiments. Namely, we obtained optimal values for $w_a = 2$, $nwa = -2$, $w_b = 20$, and deduplication threshold $t_d = 0.02$.

5.3.2 Comparison with LCS

In this experiment, we compared our sequence alignment algorithm with the standard *diff* algorithm used in Unix *diff* utility [23]. We computed the corresponding *evasion point* using both algorithms and compared their performances by computing their recall rates when varying w. The result of this experiment, shown in Figure 4, clearly shows that our proposed alignment algorithm out performs the LCS-based algorithm. This also shows that the LCS-based approach is weak. More than half of the time, the evasion locations identified using the LCS-based approach were incorrect.

In this result, we can see that with $\omega > 83$ we achieved 100% recall rate. That is, all evasion calls of the ground truth dataset are captured when $\omega > 83$. We selected a more conservative value of $\omega = 100$ for our next signature extraction experiments.

5.3.3 Evasion Signature Extraction

The next step in the extraction of evasion signatures is to build the *idf*-based filter as described in Section 4.2. For this, we obtained 119 non-evasive malware samples from the BareCloud system [17]. These are the samples that did not exhibit evasive behavior in bare metal, Anubis, Ether, and VirtualBox analysis environments. This dataset represents the non-evasive malware sample set D as described in Section 4.2. We analyzed these samples in the Anubis environment and extracted system calls, user API calls, and comparisons along with the taint dependency information. From those extracted events, we calculated the *idf* values for all observed events.

In the next step, we find the optimal *idf*-based filter threshold τ for filtering out the common execution events. As described in Section 4.2, we filter out a call event e, if $idf(e, D) < \tau$. We want a larger value of τ because we want to filter out as many common events as possible. A value of τ too small may not filter anything, and a value too large may filter out events that are part of the evasion signature. To find the optimal value for τ, we first extracted multiple evasion signatures of the ground truth samples by setting different values of τ. To compare the quality of the extracted signatures, we performed a precision-recall analysis of the extracted evasion signatures. Let Δ' be the automatically extracted signature and let Δ be the true evasion signature, which is available from the ground truth samples; the precision and recall of the evasion signature extraction is given as:

$$precision = \frac{\Delta \cap \Delta'}{\Delta'}, \quad recall = \frac{\Delta \cap \Delta'}{\Delta}.$$

Figure 5 shows the results of the precision and recall analysis. The curves represent the average characteristics of all samples. We can see that the precision decreases and recall increases as we lower the value of τ. Smaller values of τ make the *idf*-based filter weaker, and, hence, the signature includes a lot of common events, lowering its precision. We can see that the *idf*-based filter significantly increases the quality of the extracted evasion signatures if τ is selected optimally. Precision and recall rate at the crossover point, where the precision and recall curves meet, is 0.83 and the value of the threshold at this point is $\tau = 2.75$. That is,

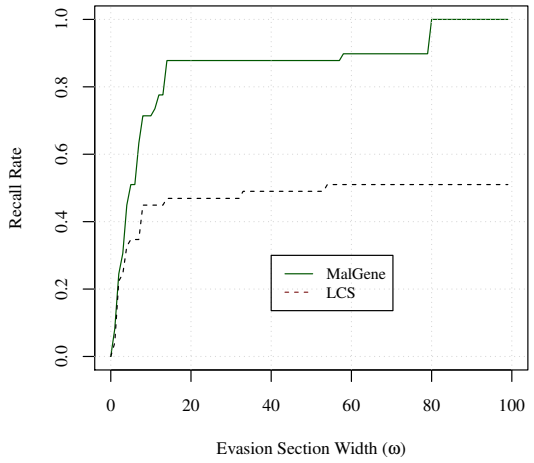

Figure 4: Comparison of *evasion gap* detection.

at $\tau = 2.75$, the algorithm is able to extract 83% of the elements of true evasion signatures with a precision of 83%. We use this value for our next experiment on cluster analysis. Figure 6 shows a sample evasion signature automatically extracted by our system from a malware sample.

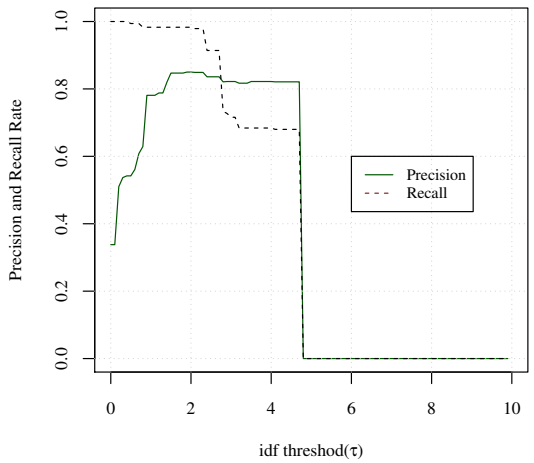

Figure 5: Precision recall analysis of the *idf* threshold τ.

5.4 Evaluation on Real-world Samples

In this experiment, we applied our approach to 2810 real-world evasive malware samples. We extracted the corresponding evasion signatures and performed hierarchical clustering as described in Section 4.4. Figure 7 shows the graphical representation of the clusters. Here, the smallest rectangles represent samples, larger rectangular patches of color represent clusters, and the shades of the color inside the patch represents the degree of similarity among individual samples within the cluster. To generate unique clusters, we cut the corresponding *dendrogram* close to the root of the tree. This way, the clusters formed are very distinct from each other, representing distinct evasion techniques. A cut at the height $h = 0.99$ produced 78 clusters. Bold lines in Figure 7 separate these clusters. A cut at the height $h = 0$ produced 1051 clusters. This represents the number of automatically extracted identical evasion signatures.

Table 1: The summary of top five clusters.

Cluster	Count	Evasion signature summary
c6	898	Exception-based emulation detection
c4	582	Cumulative timing of system calls
c5	225	Timing of exception processing
c8	172	*SystemMetrics*-based fingerprinting
c18	106	Variant of exception-based detection

```
NtOpenKey, HKLM/System/ControlSet001/Services/Disk/Enum
NtQueryValueKey, HKLM/System/ControlSet001/Services/Disk/Enum->0
CMP, NtQueryValueKey.KeyValueInformation->'Z'
CMP, NtQueryValueKey.KeyValueInformation->'wmwavboxqemu'
CMP, NtQueryValueKey.KeyValueInformation->'qemu'
```

Figure 6: A sample of an automatically extracted evasion signature (*8964683b959a9256c1d35d9a6f9aa4ef*).

We manually analyzed few samples from the top five clusters. Table 1 presents a summary of the findings.

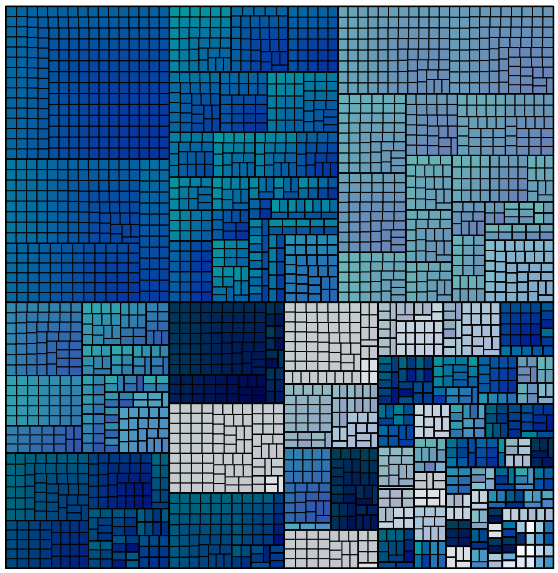

Figure 7: Hierarchical clustering of evasive malware based on their evasion signature.

6. LIMITATIONS

The main limitation of our approach is the requirement of system call sequences from both analysis environments. This limitation prevents us from using pure bare-metal execution-based malware profiles that lack system call monitoring.

One of the ways to achieve such monitoring is to use SMM-based monitoring systems, such as MALT [35].

Potentially, malware can have multiple evasion points. That is, a malware can perform multiple evasion checks at different sections of the system call sequence. Since we locate the deviation by finding the largest gap in the aligned sequence, our approach only finds one evasion point, and, as a result, extracts only one evasion signature corresponding to that evasion point. Instead of only using the largest gap, we could consider other large gaps in the aligned sequence, if any, to identify multiple evasion points. This work is limited to one evasion signature per evasive malware sample.

An adversary with the knowledge of our system can develop a mimicry attack to foil evasion signature extraction. For example, an attacker can use a Pseudo Random Number Generator (PRNG) to introduce artificial large gaps with random system calls to evade evasion point detection. However, since the malware sample evades one of the analysis systems, the actual malicious activity also causes another large gap in the sequence alignment. In this case, we could support multiple evasion points and extract separate evasion signatures from the respective evasion points. However, false positive evasion signatures, such as the one that uses PRNG techniques need to be manually identified and filtered out. The clustering of evasion signatures as described in Section 4.4 can help improve this manual process.

One of the common limitations inherent to all dynamic analysis system is the use of stalling code. A malware sample can wait for a long time before performing any malicious activity. Kolbitsch *et al.*, have proposed a technique to detect and mitigate malware stalling code [18]. Our current system will not be able extract signatures for such evasions if the stalling part of the code is deterministic, producing the same call sequence.

If a malware sample has a high level of randomization in the code execution, our approach to system call alignment may not be effective. However, if the malicious activity is long enough in one of the analysis environments, the alignment algorithm may provide an approximate location of the evasion, which can help malware analyst in further analysis. Another approach is to analyze the same malware sample multiple times in the same environment to detect and normalize such inherent randomization [20].

The proposed approach of handling sequence branching may not be effective for system call sequences produced by thread pools. This is because the order in which a thread in the thread pool is scheduled to handle callbacks can be different among instances of the malware executions.

7. RELATED WORK

7.1 Sequence Alignment

The sequence alignment problem is widely studied in bioinformatics [11, 14, 27]. Our work is based on the algorithms proposed for biological sequence alignment. We extended the algorithms to handle sequence with branches. Additionally, we adapted the algorithm and proposed optimal parameters in the context of system call sequence alignment.

Sequence alignment techniques are previously used in malware detection for finding common subsequences as signatures and for pattern matching [8,19,34]. Eskin [12] proposes a sparse sequence model to find outliers in the sequences for anomaly detection. Our use of the sequence alignment is orthogonal to those works. MalGene uses sequence alignment for identifying deviations between sequences rather than finding common patterns as signatures. Furthermore, our algorithm performs deduplication, difference pruning, and can handle branched sequences. Our approach to the extraction of evasion signature leverages data-flow dependencies to extract relevant but potentially distant events in the sequence. Data-mining techniques are used to discard irrelevant events from the evasion signature.

7.2 Differential Program Analysis

The problem of analyzing the differences between two runs of a program has been previously studied [15, 33]. The work most similar to ours is the approach of differential slicing [15]. Given a pair of two execution traces of the same program and a location of observed difference in the trace, differential slicing can identify the input difference that caused the observed difference. The main difference with our work is that the differential slicing approach requires fine-grained analysis on both analysis environments. This may not be always available in all malware analysis environments. Our approach does not require fine-grain instruction level monitoring from both analysis environments. Furthermore, to find the source of the execution difference, an analyst must first manually identify the location of the observed difference before applying the differential slicing analysis. We automate the process of identifying the location of the difference. Therefore, while previous work on differential slicing is suitable for more focused individual analysis, our approach is designed to provided an automated and practical solution to approximate program difference analysis on large scale.

7.3 Evasion Detection

Chen *et al.* proposed a detailed taxonomy of evasion techniques used by malware against dynamic analysis system [6]. Lau *et al.* employed a dynamic-static tracing technique to identify VM detection techniques. Kang *et al.* [16] proposed a scalable trace-matching algorithm to locate the point of execution diversion between two executions. The system is able to dynamically modify the execution of the whole-system emulator to defeat anti-emulation checks. Balzarotti *et al.* [4] proposed a system for detecting dynamic behavior deviation of malware by comparing behaviors between an instrumented environment and a reference host. The comparison method is based on deterministic program execution replay. That is, the malware under analysis is first executed in a reference host while recording the interaction of the malware with the operating system. Later, the execution is replayed deterministically in an analysis environment such that any deviation in the execution is an evidence of an evasion. Determinsitc replay of a malware sample may be challenging if it depends on the external network environment. In our approach, a malware can be simultaneously executed in two environments and analyzed later.

8. CONCLUSION

In this paper, we presented MalGene, a system for automatically extracting evasion signatures from evasive malware. We propose a combination of bioinformatic algorithms, data mining, and data flow analysis techniques to automate the signature extraction process, so that it can be applied to a large-scale dataset.

9. ACKNOWLEDGMENTS

We want to thank our shepherd Konrad Rieck and the anonymous reviewers for their valuable comments, and Christopher Kruegel for his insight and discussions throughout this project.

This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-13-2-4039 and by the Army Research Office (ARO) under grant W911NF-09-1-0553. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

10. REFERENCES

[1] Anubis. `http://anubis.cs.ucsb.edu`.

[2] Intel Virtualization Technology. `http://www.intel.com/technology/virtualization/`.

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, 1997.

[4] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, G. Vigna, and S. Antipolis. Efficient Detection of Split Personalities in Malware. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[5] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[6] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Dependable Systems and Networks With FTCS and DCC*, 2008.

[7] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *IEEE Symposium on Security and Privacy*, 2010.

[8] S. E. Coull and B. K. Szymanski. Sequence alignment for masquerade detection. *Computational Statistics & Data Analysis*, 52(8):4116–4131, 2008.

[9] M. O. Dayhoff and R. M. Schwartz. A model of evolutionary change in proteins. In *Atlas of Protein Sequence and Structure*, 1978.

[10] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] R. C. Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 2004.

[12] E. Eskin. *Sparse sequence modeling with applications to computational biology and intrusion detection*. PhD thesis, 2002.

[13] P. Ferrie. Attacks on virtual machine emulators. Technical report, Symantec Corporation, 2007.

[14] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*.

[15] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *IEEE Symposium on Security and Privacy*, 2011.

[16] M. Kang, H. Yin, and S. Hanna. Emulating emulation-resistant malware. *ACM workshop on Virtual machine security. ACM*, 2009.

[17] D. Kirat, G. Vigna, and C. Kruegel. BareCloud: bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium (USENIX)*, 2014.

[18] C. Kolbitsch, E. Kirda, and C. Kruegel. The Power of Procrastination: Detection and Mitigation of Execution-stalling Malicious Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[19] V. Kumar, S. K. Mishra, and L. Bhopal. Detection of malware by using sequence alignment strategy and data mining techniques. *International Journal of Computer Applications*, 62(22), 2013.

[20] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting Environment-Sensitive Malware. *Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 338–357, 2011.

[21] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. 2008.

[22] D. W. Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004.

[23] E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1986.

[24] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 1970.

[25] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[26] G. Pék. nEther : In-guest Detection of Out-of-the-guest Malware Analyzers. *Proceedings of the Fourth European Workshop on System Security. ACM*, 2011.

[27] V. Polyanovsky, M. A. Roytberg, and V. G. Tumanyan. Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences. *Algorithms for Molecular Biology*, 2011.

[28] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. *Information Security*, pages 1–18, 2007.

[29] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction, 2004.

[30] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 1981.

[31] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[32] A. J. Vakharia and U. Wemmerlöv. A comparative investigation of hierarchical clustering techniques and dissimilarity measures applied to the cell formation problem. *Journal of operations management*, 1995.

[33] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *Symposium on Software Testing and Analysis*, 2010.

[34] A. Wespi, M. Dacier, and H. Debar. *An intrusion-detection system based on the Teiresias pattern-discovery algorithm*. IBM Thomas J. Watson Research Division, 1999.

[35] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. Using hardware features for increased debugging transparency. In *IEEE Symposium on Security and Privacy*, May 2015.