MAXS: Scaling Malware Execution with Sequential Multi-Hypothesis Testing

Phani Vadrevu
University of Georgia
Athens, Georgia 30602
vadrevu@cs.uga.edu

Roberto Perdisci
University of Georgia
Athens, Georgia 30602
perdisci@cs.uga.edu

ABSTRACT

In an attempt to coerce useful information about the behavior of new malware families, threat analysts commonly force newly collected malicious software samples to run within a sandboxed environment. The main goal is to gather intelligence that can later be leveraged to detect and enumerate new malware infections within a network. Currently, most analysis environments "blindly" execute each newly collected malware sample for a predetermined amount of time (e.g., four to five minutes). However, a large majority of malware samples that are forced through sandbox execution are simply repackaged versions of previously seen (and already analyzed) malware. Consequently, a significant amount of time may be wasted in analyzing samples that do not generate new intelligence.

In this paper, we propose MAXS, a novel probabilistic multi-hypothesis testing framework for scaling execution in malware analysis environments, including bare-metal execution environments. Our main goal is to automatically recognize whether a malware sample that is undergoing dynamic analysis has likely been seen before (e.g., in a "differently packed" form), and determine if we could therefore stop its execution early while avoiding loss of valuable malware intelligence (e.g., without missing DNS queries to never-before-seen malware command-and-control domains).

We have tested our prototype implementation of MAXS over two large collections of malware execution traces obtained from two distinct production-level analysis environments. Our experimental results show that using MAXS we are able to reduce malware execution time by up to 50% in average, with less than 0.3% information loss. This roughly translates into the ability to double the capacity of malware sandbox environments, thus significantly optimizing the resources dedicated to malware execution and analysis. Our results are particularly important for bare-metal execution environments, in which it is not easy to leverage the economies of scale that characterize virtual-machine or emulation based malware sandboxes. For example, MAXS could be used to significantly cut the cost of bare-metal analysis environments by reducing the hardware resources needed to analyze a predetermined daily number of new malware samples.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
© 2016 ACM. ISBN 978-1-4503-4233-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897873

1. INTRODUCTION

The common practice of studying malicious software (or *malware*) in a controlled environment is analogous to *in vitro* studies of biological pathogens, which allow scientists to understand how a pathogen attacks the victim system and what mechanisms may be used to develop a cure. For example, malware analysts routinely force newly collected samples to run within a sandboxed environment, and study their system- and/or network-level behavior to derive possible defense mechanisms [1, 2, 8, 9].

Unfortunately, the *in vitro* behavior of a malware sample may differ significantly from what may be observed *in vivo*, namely on a *live* infected machine. One of the leading causes of such a *behavior divergence* is the fact that the malware may be able to detect the nature of the execution environment itself, and alter the behavior if it believes it is being studied [4]. For instance, a malware sample may attempt to detect the side-effects imposed by a virtual-machine (VM) or emulation-based analysis environment, thus indicating whether the system has been heavily instrumented to record detailed information about how the malware "exploits" or infects the victim machine.

One possible approach towards mitigating such drawbacks is the use of *bare-metal* malware execution [15, 16], in which the malware is forced to run on a "native" system with no virtualization, emulation, or other heavy system instrumentations, thus bringing us closer to *in vivo* malware analysis. While bare-metal execution can only provide less fine-grained information regarding system-level operations performed by the malware, compared to emulation-based systems, it still allows for *transparently* collecting a range of interesting behaviors, including detailed network activities that can be leveraged for defense purposes. Unfortunately, bare-metal analysis environments can incur much *higher hardware costs*, because they cannot as easily benefit from the economies of scale that are instead leveraged by traditional VM- or emulation-based analysis environments, thus limiting the number of malware samples that can be analyzed per day.

To reduce the cost of bare metal execution, in this paper we propose MAXS[1], a probabilistic multi-hypothesis testing

[1]Malware Analysis eXecution Scaler

framework that aims to increase the *capacity* of malware analysis environments, including bare-metal environments. Specifically, MAXS aims to significantly boost (e.g., double) the average number of malware samples that can be analyzed per day by a given malware analysis system. We accomplish this goal by dynamically optimizing the amount of execution time dedicated to the analysis of each malware sample. To this end, we develop a new probabilistic decision framework that operates *sequentially* on malware behaviors (e.g., malware-generated network events) observed during malware analysis.

Our work is based on the following observations:

1. Many (if not most) newly collected malware samples are simply "re-packed," obfuscated variants of previously analyzed malware families [11,13,21,23].

2. While appearing syntactically unique, a new sample may behave in a way very similar or identical to samples we have already analyzed, thus producing little or no new actionable intelligence (e.g., new malware-control domains to be blacklisted).

3. Malware analysis environments typically execute each sample "blindly" for a predetermined amount of time (e.g., four to five minutes) [5,7], irrespective of the behavior exhibited by the sample being tested and of malware observed in the past.

MAXS improves the scalability of a malware analysis environment by dynamically (and probabilistically) deciding whether a sample really needs to run for a predetermined maximum execution time, or if its execution should instead be preemptively stopped. This decision is made by incrementally analyzing the (partial) behavior exhibited by a sample, and by continuously comparing it with the behavior profile of a large set of previously observed malware families. Specifically, as a malware sample is forced to execute, it will likely start generating a number of network *events*. For instance, the sample may query one or more domain names, initiate TCP flows towards outside IP addresses, send HTTP requests, etc. Every time a new event is observed, MAXS takes a look at the current execution trace and computes two quantities: (1) the probability, P_f, that the sample belongs to a previously analyzed malware family; and (2) the probability, P_b, that if we continue to run it, the current sample will produce previously unseen malware behaviors (e.g., queries to never-before-seen malware-control domains, which could be added to a blacklist). Intuitively, if P_f is high and P_b is very low, we can preemptively stop the sample's execution while *minimizing the risk of information loss.*

MAXS is different from previous work, such as [5]. In particular, [5] requires access to fine-grained system-level information and therefore cannot be directly applied to bare-metal execution environments. In contrast, MAXS is a *generic* probabilistic decision framework that can be applied with little or no changes to a variety of malware analysis outputs, including network activities produced by bare-metal execution environments, system-level information produced by VM- or emulation-based environments, as well as static malware code analysis information (see Section 4.4). In addition, whereas [5] attempts to decide whether to preemptively stop the execution of a malware sample using a "one shot" decision test at a pre-selected fixed time (e.g., after sixty seconds of execution), MAXS provides a *sequential testing* framework that allows us to *dynamically* decide if/when a sample's execution should be stopped. Finally, unlike [5], MAXS explicitly embeds into its probabilistic decision framework the loss of information (e.g., unobserved malware-control domain names) caused by a possibly premature termination of a sample's execution, and thus aims to optimize the trade-off between resource savings and information loss for each malware family.

In summary, we make the following main contributions:

- We propose MAXS, a novel probabilistic decision framework that aims to boost (e.g., double) the capacity of malware analysis environments, including bare-metal execution environments.

- We provide a detailed description of our proposed probabilistic framework, which is based on multi-hypothesis sequential probability ratio tests, and discuss how MAXS can be used to improve the scalability of malware analysis systems while minimizing information loss.

- We evaluate MAXS over two large datasets of malware execution traces produced by two different production-level malware analysis environments, and show that in average we can decrease the time needed to analyze these malware samples by up to 50% with less than 0.3% information loss.

2. MAXS FRAMEWORK OVERVIEW

In this section, we provide a brief overview of how our MAXS framework works. More technical details on our approach will be presented in Section 3.

Background. A typical production-level malware analysis environment may process tens or even hundreds of thousands "new" malware samples per day. Each malware sample is usually "blindly" executed for a fixed time window (e.g., 4 to 5 minutes), during which the malware behavior (system and/or network level) is recorded. For instance, as the malware sample is executed, it may query a set of malware-control domain names. These domains (or the resolved IP addresses) may then be added to a domain name blacklist, and used to enumerate machines infected with the same type of malware in the wild (e.g., in live ISP or enterprise networks) [3,6].

Goals and Benefits. MAXS aims to reduce the amount of execution time dedicated to analyzing each malware sample, instead of executing each sample for a fixed time. As discussed in Section 1, the main intuition is that many of the (supposedly) new samples are in fact repacked, obfuscated versions of previously seen and analyzed malware. Therefore, our goal is to identify such cases, and preemptively stop malware execution if we believe the sample is highly unlikely to produce new actionable intelligence (e.g., new malware-control domains that can be added to a domain blacklist).

Effectively, MAXS aims to significantly reduce the average execution time that needs to be dedicated to each malware sample (e.g., from 4 to 2 minutes). This savings can be leveraged in multiple ways:

- MAXS can help decrease the cost of malware analysis systems, because fewer resources are needed to handle a predetermined daily volume of malware samples. For instance, we could design and provision a new bare-metal malware execution environment at a lower overall hardware cost.

- Given an existing malware analysis system, MAXS can help to significantly increase (e.g., double) its capacity, measured as the average number of malware samples that can be processed per day.

- In alternative, an existing malware analysis system could spend less time re-analyzing previously seen malware families, and reallocate the time saved thanks to MAXS to running *truly new* samples (e.g., samples belonging to new or underrepresented malware families) for longer periods of time, thus potentially gaining more insight into their malicious behavior.

Approach. MAXS works in two phases: (1) a *learning phase*, in which malware samples are grouped into families and behavior profiles are derived; and (2) an *operational phase*, in which the events generated by the execution of new malware samples (e.g., new domain name queries) are *sequentially tested* against the previously learned family behavior profiles, to decide whether we should preemptively stop a sample's execution.

Figure 1a provides an overview of the learning phase. Given an initial set of malware samples and the respective execution traces (e.g., obtained by running the samples for a fixed amount of time), we first need to cluster similar samples into *families*, for example by using techniques similar to [14, 18, 22]. Essentially, a malware family is a group of malware samples that produce similar *events*, such as similar network activities (e.g., similar DNS queries, HTTP requests, TCP flows, etc.). Notice that throughout the reset of the paper we intentionally use the generic term "*events*" because MAXS is a generic framework that can be applied to different types of malware information, including network events, system events, and also information extracted using static analysis (see Section 4.4).

Once the malware families are formed, from each family we derive a family behavior profile that summarizes the set of events generated by all the samples in the family. These behavior profiles are used during the operational phase to determine if a new malware sample likely belongs to a previously seen family, and to dynamically decide if we should continue executing the sample or not.

The learning phase we just described can be performed periodically (e.g., once a week) offline, to capture the natural evolution of existing malware families and to discover previously unseen groups. As the new family profiles become available, we can replace the older ones without having to pause the malware analysis infrastructure.

Figure 1b shows how MAXS works during the operational phase. Let m be a new malware sample to be executed. Every time m generates a new *event* (e.g., a new DNS query) we compare the events produced so far to each of the family behavior profiles derived during the learning phase. Specifically, we compute the probability, $P_{f_i}(t)$, that the sample belongs to family f_i, given all events generated by m until the current execution time t. If $P_{max}(t) = \max_i\{P_{f_i}(t)\}$ is greater than a tunable threshold β (which can be computed during the learning phase), we assign the sample to the family f_k whose $P_{f_k}(t) = P_{max}(t)$. Otherwise, we wait for the next event and repeat the process.

After the sample has been assigned to a family, say f_k, at a given execution time, say t_k, we compute the probability, $P_{b_k}(t)$, that a malware samples belonging to f_k will produce previously unseen events at any time $t \geq t_k$. Specifically, let T_{exe} be a predetermined maximum execution time allowed by the analysis infrastructure, and t_e be the time of the latest event of interest generated by the currently running sample m. If $P_{b_k}(t') < \gamma$ for each $t' \in (t_e, T_{exe}]$, we preemptively stop the execution of the sample at time t_e. Otherwise, we continue executing m, and repeat the process described above as soon as a new event is generated.

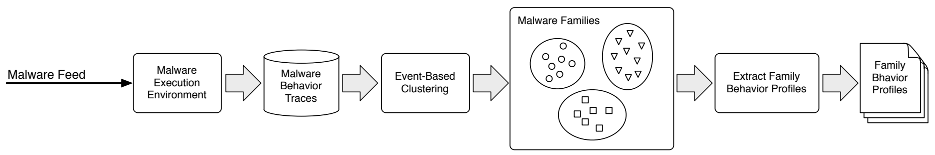
Intuitively, we preemptively stop execution if it is highly unlikely that the sample will generate any never-before-seen events after t_e. If the above condition is never satisfied, we keep executing the sample up to T_{exe}.

Resource Savings v.s. Information Loss. The approach outlined above aims to reduce the amount of time spent on executing malware samples that will produce no new *actionable intelligence*. For instance, assume our goal is to build a blacklist of malware command-and-control (or C&C) domain names. In this case, the actionable intelligence is represented by DNS queries to previously unseen C&C domains generated while a sample runs within the malware execution environment. If we executed a sample for the maximum time T_{exe} and the sample produced no new domain names (i.e., no domain is queried, or all queried domains were already queried by previously analyzed malware samples), we would essentially waste the entire T_{exe} time slot. Conversely, if at time $t < T_{exe}$ we decided that the sample is unlikely to produce new actionable intelligence and stopped its execution, we would be able to save $(T_{exe}-t)$ time that could be allocated to executing other samples, thus increasing the capacity of the malware execution infrastructure.

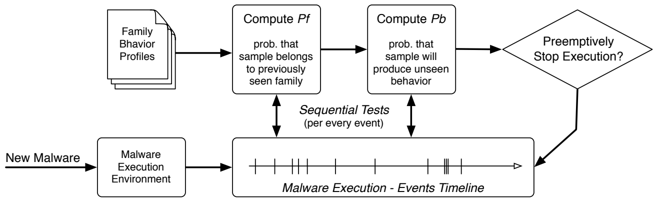
Naturally, there exists a tension between resource savings and behavior observation completeness. As explained earlier, MAXS makes decisions in a probabilistic way. For example, assume that we preemptively stop a sample's execution at time $t < T_{exe}$. It is indeed possible that if we had let the sample run, it would have produced new actionable intelligence (e.g., new C&C domains) during the $[t, T_{exe}]$ execution interval. In this case, we would experience *information loss.* To minimize such cases, MAXS explicitly aims to optimize the trade-off between resource savings and information loss by embedding both concepts into the learning of the parameters β and γ, which influence the attribution of a sample to a malware family and how early execution should be halted, respectively (see Section 3 for further details).

3. MAXS FRAMEWORK DETAILS

In this section we provide a more detailed description of our technical approach. As mentioned earlier, MAXS is a *generic* framework that can be applied to a variety of definitions of malware-generated *events*, including network activities such as DNS requests, HTTP traffic, TCP flows, etc., and even static analysis information (see Section 4.4). In addition MAXS could be extended to using system-level events, such as file or registry changes. Therefore, in the following we will primarily discuss our MAXS framework in terms of generic *events*, and provide only some examples of concrete event definitions to make this section more readable. We will present a concrete application of MAXS to real-world malware analysis environments in Section 4.



(a) Learning phase: malware family discovery and family behavior profile extraction.


Family Bhavior Profiles
Compute *Pf*
prob. that sample belongs to previously seen family
Compute *Pb*
prob. that sample will produce unseen behavior
Preemptively Stop Execution?
Sequential Tests
(per every event)
New Malware
Malware Execution Environment
Malware Execution - Events Timeline

(b) Operational phase: overview of sequential tests applied to new malware samples.

Figure 1: MAXS framework.

3.1 Learning the Family Profiles

We now explain how to learn the malware family behavior profiles (see Figure 1a). This task is divided into two main parts: (a) clustering malware samples into families based on the events they generate during execution, and (b) deriving a *behavior profile* for each family, which summarizes the events generated by all the samples in a family. This learning process can be applied periodically (e.g., once a week), to update the family behavior profiles to the latest malware trends.

Malware Clustering. Because MAXS aims to be a generic framework, it does not dictate what similarity functions and clustering algorithm should be used to partition a set of malware samples into families. The main contributions of this paper are on building MAXS's probabilistic framework, and not on the specific algorithms used to cluster malware samples. Therefore, methods similar to previously proposed malware clustering systems, such as [14, 18, 22], could be applied for this task.

As discussed more in detail in Section 4, we concretely apply MAXS to network events, such as sequences of DNS queries, which are available as the output of bare-metal malware execution environments. In the particular application of MAXS to DNS query information, we measure similarity between two malware samples by computing the Jaccard index over the sets of domain names queried by the two samples. We then use the DBSCAN clustering algorithm [10] over the Jaccard-based similarity matrix to partition a malware dataset into malware families.

Measuring Behavior Variability. Once the malware samples are grouped into families, we focus on the behavior of all the samples in a given family. In particular, we want to model how *variable* the behavior of the malware family is. To better explain what we mean with *variable*, let us use an example. Assume the malware-generated *events* we are interested in are all the DNS queries issued by a sample during execution. Given a malware family, f_k, if the vast majority of all samples belonging to f_k query the same exact set of domain names, in the same sequential order, we can say that the behavior of samples in f_k is very consistent, and therefore has *low variability*. Conversely, if each of the samples in f_k queries several domain names that have never been queried by any of the other malware samples in f_k, we can say that malware belonging to f_k exhibit *highly variable* behavior.

We now define how MAXS models event variability for each malware family. Let us consider a given family, f_k, and let $e_1^{(j)}, e_2^{(j)}, \dots, e_{n_j}^{(j)}$ be the sequence of n_j events generated by the j-th malware sample, $m_j \in f_k$. Now, let E_I be the list of events generated by all malware in f_k at event sequence index I. Namely, $E_I = \{e_I^{(1)}, \dots, e_I^{(|f_k|)}\}$, where $|f_k|$ is the number of samples in family f_k. Notice that if a given malware sample m_j generates less than I events, its $e_I^{(j)}$ will be represented with a special *null* event.

We then define a function $S(E_I)$, which returns the set of *distinct events* in list E_I, and a function $C(e, E_I)$ that counts how many times an event e appears in the list E_I. Using the above notation, we define the variability of family f_k at event sequence index I as the *normalized entropy* of the events in E_I (notice that $\frac{C(e,E_I)}{|f_k|}$ estimates the probability of observing event e at index I):

$$V_I = \frac{1}{log(|f_k|)} \sum_{e \in S(E_I)} - log\left(\frac{C(e, E_I)}{|f_k|}\right) \frac{C(e, E_I)}{|f_k|} \tag{1}$$

Intuitively, if all the events in E_I are the same, namely all malware in the family generate the very same event at sequence index I, then $V_I = 0$ (no variability). Conversely, if each malware generates a different event, then $V_I = 1$ (i.e., max variability).

Family Event Likelihood. Now that we have defined behavior variability for a malware family, say f_k, we can also compute the likelihood that an event e may be generated by a member of f_k. Specifically, if e has already been observed in past samples belonging to f_k, we can approximate the likelihood that a future sample belonging to f_k will generate the same event e by computing e's relative occurrence frequencies, compared to all other events generated by the same family. On the other hand, if e has never been observed before (i.e., no previously observed sample in f_k generated e), we still cannot exclude that a malware sample in f_k could generate that event. To take this into account, we compute e's likelihood by leveraging the variability measure defined earlier. Essentially, the more variable the family's behavior, the higher the likelihood that a new sample of that family may generate a previously unseen event e.

To formalize what we described above, we define the likelihood that an event e may be generated by a malware sample in family f_k at event sequence index I as follows:

$$L_I(e) = \begin{cases} \alpha + (1-\alpha)\dfrac{C(e, E_*)}{|f_k|} & \forall e : C(e, E_*) > 0 \\ \alpha V_I & \forall e : C(e, E_*) = 0 \end{cases} \tag{2}$$

where α is a constant set to 0.5 in our experiments, E_* is the list of events generated by all malware in f_k at any event sequence index, and $C(e, E_*)$ counts the number of samples that generated event e at least once (e.g., if e was a domain name query, $C(e, E_*)$ would count how many samples queried e at least once).

Notice that if e has never been observed (i.e., $C(e, E_*) = 0$) in samples belonging to f_k, then its likelihood $L_I(e) \in [0, \alpha]$. Notice also that because V_I is a function only of the index I, any event e for which $C(e, E_*) = 0$ will be assigned the same exact likelihood value. Conversely, if $C(e, E_*) > 0$, then $L_I(e) \in (\alpha, 1]$. In other words, if e had been generated at least once by a previously observed sample assigned to f_k during the learning phase, then its likelihood will be higher than the likelihood of any previously unseen events (because $L_I(e) > \alpha$, according to Equation (2)).

Reducing Clustering Noise using Event Likelihood. Malware clustering is by nature an imperfect task [22], resulting in potential noise (i.e., incorrectly grouped samples) within the malware families we learn. To alleviate this problem, we leverage the event likelihood defined in Equation (2) for denoising purposes, thus improving the learning phase. We proceed as follows, using a greedy algorithm.

Given a malware family f_k, we randomly select a sample $m \in f_k$. Then, using only the remaining samples set, $f_k^{'} = f_k - m$, we learn the behavior variability (Eq.(1)) for each event sequence index I. Finally, we compute the likelihood of each event generated by m (i.e., we treat m as if it was a future sample of family $f_k^{'}$), and remove m from family f_k if any of these likelihoods is equal to zero. More formally, we remove m if $\exists e_I^{(m)} : L_I(e_I^{(m)}) = 0$, where $e_I^{(m)}$ is the I-th event generated by m.

Intuitively, if a malware sample generates an previously unseen event e at index I, whereas all other samples in f_k had generated a *constant* (zero variability) event $e' \neq e$, this is an indication that m may not belong to f_k after all. Therefore, we reverse our previous decision (made during the malware clustering phase) that m belongs to f_k, and thus remove it from that set. Notice, however, that our requirement for removing a sample is quite conservative, because we require that $L_I(e_I^{(m)}) = 0$. Also, while this denoising process is greedy, and may in some corner cases erroneously remove a sample from its correct family assignment, this does not cause significant problems because slightly reducing the size of a malware family cluster will still allow us to learn reliable behavior profiles from the remaining family samples. At the same time, removing truly noisy instances will improve the accuracy of the profiles.

Minimum Cumulative Event Likelihood. After the denoising process terminates, thus giving us the final set of samples belonging to family f_k, we also compute the family's *minimum cumulative event likelihood* (MCL) for each possible event sequence index I. Specifically,

$$MCL(f_k, I) = \min_{m \in f_k} \prod_{i=1 \ldots I} L_i(e_i^{(m)}) \tag{3}$$

where $e_i^{(m)}$ is the i-th even generated by a malware $m \in f_k$. The MCL will be useful later to verify if we should preemptively stop malware execution or not.

3.2 Deciding *If* and *When* to Stop Malware Execution

Given a new, never-before-seen malware sample, we can leverage the family behavior profiles defined in Section 3.1 to answer the following questions: Does the new sample belong to a *previously learned* malware family? If yes, how variable is the behavior of the samples in that family? What is the probability that samples in the considered family will generate previously unseen events after a given event sequence index?

Intuitively, if the new sample closely matches a previously learned family profile, and if we know that all samples in that family have extremely low behavior variability (as defined in Section 3.1), it is likely that the new sample will exhibit the same previously observed behavior. As such, the new sample is unlikely to generate new actionable malware intelligence, and we can therefore preemptively stop its execution while causing little or no information loss. We formalize these concepts below.

Does the new sample belong to a previously learned malware family?
Assume we have already collected and executed a set of malware samples, from which we have learned a set of malware families and related behavior profiles, as described in Section 3.1 (notice that the learning process can be run periodically to update the profiles). Now, let m be a new, never-before-seen malware sample that is fed to the malware analysis infrastructure. At every event generated by m, e.g., for each domain name queried by m while running in a bare-metal environment, we compute the probability that m belongs to one of the previously learned malware families.

More formally, let f_k represent the k-th out of N malware families. Also, let e_j be the j-th event generated by m while it is being analyzed, and $L_j^{(k)}(e_j)$ be the likelihood that e_j is also generated by malware samples in f_k at event sequence index j, as defined in Equation 2. To decide *if* and *which* malware family m belongs to, we use a *multi-hypothesis sequential probability ratio test* (MSPRT) as follows:

$$P_{f_k}(m, J) = \frac{\prod_{j=1}^{J} L_j^{(k)}(e_j)}{\sum_{\nu=1}^{N} \left(\prod_{j=1}^{J} L_j^{(\nu)}(e_j) \right)} \tag{4}$$

where $P_{f_k}(m, J)$ is the probability that m belongs to f_k, computed after m has generated J events (e.g., after J domain names are queried).

As a concrete example, assume we are interested in domain name query events. Say that we have run m for a time t, and during this time m queried two domain names, e_1 =`mybots1.cc` and e_2 =`mybots2.cc`. Then, $L_1^{(3)}(e_1)$ would represent the probability that the first DNS query issued by a malware samples in family f_3 is a query to `mybots1.cc`, and $L_1^{(3)}(e_2)$ is the probability that the second domain queried by the samples in f_3 is `mybots2.cc`. Essentially, while m is running, for every new event of interest it generates we compute a new value for the probability P_{f_k} that takes into account the probability of all events generated so far by m.

For each new event generated by m, we check the following: if there exists a family f_k^* for which $|f_k^*| > \theta$ and $P_{f_k^*} > \beta$, where β and θ are tunable parameters, we say that m belongs to family f_k^*. The parameter θ is used to make sure that f_k^* contains enough samples so that we can be confident about the behavior profile we learned for that family. This is important because if f_k^* contains very few samples, its behavior is learned over few data points and may not allow us to reliably compute $P_{f_k^*}$. On the other hand, parameter β determines how "confident" we need to be before we assign m to any of the N possible malware families. This also means that m may never be assigned to any family, and be therefore executed for the maximum allowed time, T_{exe} (e.g., in those cases when belongs to a new malware strain).

Should we preemptively stop execution?
Assume that malware sample m, after generating J events of interest (e.g., J DNS queries), is assigned to family f_{k^*}. We now need to determine if we should keep running m or not. To this end, we look back at its family behavior profile and compute the probability that malware samples in f_{k^*} will generate previously unseen events after event sequence index J.

More formally, using Equation 2 we compute the likelihood $L_x(e)$ of a hypothetical new event e, which was never observed during the learning phase for family f_{k^*}, for every event sequence index $x = (J+1), (J+2), \ldots$, up to the maximum number of events generated be any of the samples in the family. Then, we compute the following probability:

$$P_b(J) = \max_{x>J}\{L_x(e)\} \tag{5}$$

If $P_b(J) < \gamma$, where γ is a small tunable threshold, we determine that there is a low probability (less than γ) that, if we keep running it, m will generate an event never seen during the learning of the behavior profile for its family f_{k^*}, and therefore we can stop its execution. Intuitively, the parameter γ allows us to control the *trade-off between the execution time savings and the amount of information loss* we can afford to tolerate.

If we decided to keep running m, for every new event it generates we recompute the probability P_b. As soon as we find an event sequence index $H \geq J$ for which $P_b(H) < \gamma$, we preemptively stop executing m. Otherwise, we keep running m for the entire maximum amount of time permitted by the malware execution infrastructure.

There is one more fine detail that we consider, before stopping the execution of m: we double check that its cumulative event likelihood is greater than the family's MCL (see Equation 3). Formally, we stop m only if $\prod_{i=1\ldots H} L_i(e_i^{(m)}) >= MCL(f_k, H)$. Essentially, we want to make sure, before we stop the sample's execution, that it is very unlikely that we are making a mistake by assigning the sample to family f_k and stopping its execution at event index H.

4. EVALUATION

In this section, we describe the set of experiments we have performed to demonstrate how MAXS can be used to save malware execution resources with only a small loss of malware intelligence.

4.1 Datasets

To evaluate our MAXS framework, we use two large malware datasets obtained from two different production-level malware execution and analysis systems operated by two different organizations (a security company and a large academic research institute). In the following we refer to these production malware analysis systems as S_A and S_B. The first dataset was collected from S_A and contains 3,581,503 malware samples that were received and analyzed across several weeks between July and December 2013. The second dataset, collected from S_B, contains 840,422 malware samples received within 6 consecutive days during November 2014. In addition to the actual malware executable files, we also collected all network traffic generated by each sample when executed by the respective analysis system from which they were collected.

Overall, we noticed that about 65% and 52.4% of these samples from S_A and S_B, respectively, did not exhibit any meaningful network activity when executed. This could be explained by the fact that both S_A and S_B use a VM-based analysis environment, and are therefore susceptible to *anti-VM* tactics. Consequently, if the presence of the analysis environment is detected (or suspected), a malware sample may decide to alter its behavior and avoid exposing malicious network activities.

Because we mainly focus on studying the effects of MAXS on *bare-metal malware execution*, in the following we only consider malware samples that exhibit at least some network activities (e.g., at least one domain name query or network flow). The key rationale is that samples that stop executing (or change behavior) due to the detection of a VM or emulation based environments would more likely run as expected in a bare-metal environment, and likely generate some network activities needed to monetize the malware's malicious behavior.

We are therefore left with 1,251,865 malware samples from S_A, and 400,041 samples from S_B. In the following, we will refer to these two malware datasets and the related network traces as M_A and M_B, respectively. Table 1 summarizes the properties of the two datasets. Notice that, like most other malware analysis systems, both S_A and S_B force each sample to run "blindly" for a prefixed amount of time: 240 and 360 seconds, respectively. The samples were collected across a period of 77 days from S_A and across 6 consecutive days from S_B. To obtain dataset M_A, in average we collected 16,258 malware samples (and related network traces) per day. Of these, 15,431 performed at least one domain name query while running (similar observations can be made for dataset M_B), as indicated in Table 1. The reason why we report the number of samples that generate some DNS traffic is that in the following we describe an application of our MAXS framework in which each *event* used to compute the probabilities described in Section 3 is represented by a different domain name query.

dataset	prefixed run time	collection days	avg. samples / day	avg. samples with DNS queries / day
M_A	240s	77	16,258	15,431
M_B	360s	6	66,674	62,063

Table 1: Summary of malware dataset properties.

4.2 Experiments Setup

As we explained in Section 2 and 3, MAXS is a generic framework that can be applied to different types of *events*. However, in this paper, our main focus is on optimizing the resources used by bare-metal malware execution environments. For our evaluation, we therefore consider malware-generated network activities, which can be easily collected as the malware runs in bare-metal environments. Specifically, we focus on domain name queries (Section 4.3), because extracting malware domains from malware-generated traffic is one of the main ways for *seeding malware domain blacklists*. Therefore, in this application of MAXS, an *event* is represented by a domain name query (more details are presented in Section 4.3).

Additionally, we apply our MAXS framework to malware information extracted via static analysis (Section 4.4). In this second application, our goal is to find malware samples that we do not even need to start running, because MAXS tells us that they will (with high probability) generate no new malware intelligence (e.g., no previously unseen malware domains). In this case, an *event* is represented by a set of static analysis features (explained in more details in Section 4.4). Finally, we combine the application of MAXS to both static analysis information and network activities, to show the overall resource savings that can be achieved using our framework (Section 4.5).

4.2.1 Measuring Time Savings.

In our evaluation, we measure the amount of execution time saved by using MAXS, compared to the time needed by the two production-level malware execution analysis environments S_A and S_B. For example, let m be a malware sample, and T be the fixed execution time originally assigned to m by an analysis system S. Also, let $T_M(m)$ be the amount of time for which m would actually be executed before MAXS decides to preemptively stop running it. This time $T_M(m)$ includes the time to set-up the bare metal environment, which can be reduced to less than 5 seconds as was shown in [15].

For each sample, we measure $(T - T_M(m))/T$ as the time savings. We repeat the same measurement for each malware sample in both datasets M_A and M_B, and report the overall time savings as a percentage of the total time that would be necessary to run all samples without applying MAXS. Notice that for malware in M_A, T is 240 sec, whereas T is 360 sec for malware in M_B. If MAXS decides not to preemptively stop execution of a sample, we assume the sample is allowed to run up to the maximum time T. Notice also that in the case of the application of MAXS to static analysis information (Section 4.4), $T_M(m)$ can be equal to zero, if MAXS decides not to run sample m at all.

4.2.2 Measuring Information Loss.

Naturally, while we aim to save execution time, we would like to do so by minimizing the amount of malware intelligence lost by dynamically shortening the average amount of time dedicated to running each sample. To this end, we measure information loss as the overall number of malware domain names that we miss to observe due to MAXS's decision of preemptively stopping the execution of malware samples, compared to if we had let the samples run for the maximum allowed time. For example, let M be a set of malware samples, D_{train} be the set of domains queried by the training samples (i.e., the samples used to build the family behavior profiles), and D_{new} be the overall set of distinct *new* domains (not seen in the training phase) queried by running each sample in M for the maximum execution time T allowed by the underlying analysis environment, S. Now, assume that, by applying MAXS, the execution of a subset of malware samples $M' \subset M$ is stopped earlier than T, and that this causes us to miss a subset $D' \subset D_{new}$ of domains (i.e., the domains in D' are not queried by any of the malware samples that run through MAXS). Then, we measure the information loss as $|D'|/|D_{new}|$.

4.3 Experiment 1: Malware Domain Intelligence

In this set of experiments, MAXS is setup to continuously monitor the sequence of domain name queries generated while a malware sample runs. As explained in Section 3, at every event (i.e., at each DNS query in this particular application), MAXS first computes the probability that a sample belongs to a previously learned malware family. Then, once a family assignment is made, we leverage information about past family behavior to decide when (and if) execution should be preemptively stopped. We then measure time savings and the related information loss, as defined in Section 4.2.

To perform the family learning phase (see Section 2), we used the DBSCAN clustering algorithm[2]. We measure the similarity between two malware samples by simply computing the Jaccard index between the sets of domain names queried by the two samples. Notice that our focus here is not to propose new advanced malware clustering methods. Rather, we leverage simple well known clustering approaches to perform the learning phase needed to bootstrap our sequential probabilistic decision process.

We performed experiments on both datasets M_A and M_B. To conduct this particular set of experiments and compute the time savings and information loss, we only considered samples that exhibited at least one DNS query, which represent more than 93% of the samples in both datasets (see Table 1). In Section 4.4, we will expand our evaluation to considering any malware sample, regardless of whether they issue any DNS query.

4.3.1 Parameter Selection

As already mentioned in Section 3, in order to use MAXS, we need to set two main parameters: (1) β, which affects how soon a malware sample can be assigned to a family; and (2)

[2]We empirically set the algorithms parameters $eps = 0.4$ and $MinPts = 3$, via pilot experiments

γ, which affects how soon the system will stop running a malware sample after it has assigned the sample to a cluster.

To measure how changes in these parameters affect MAXS's performance, we conducted a set of tests using 100 different value pairs for β and γ. For these parameter search experiments, we limited ourselves to using a small portion of dataset M_A consisting of malware samples collected during four contiguous days in July 2013. We used the first three days for training and applied MAXS to the malware from the fourth day as a validation set. Figure 2 reports the results of these train-test experiments.

We can see that as β increases for a given value of γ, the time savings decrease noticeably. This is expected because as the family assignment threshold is made stricter, we will need to wait for more events to take place before we can be confident enough (above the β threshold) about assigning the sample to a given family. Also, we can notice that the lower γ, the lower the time savings, because we are asking MAXS to be highly confident about the fact that the sample will not query unseen domain names, before we preemptively stop its execution.

By setting $\beta = 0.05$ and $\gamma = 0.1$, we are able to reach time savings above 40% with less than 0.1% of samples causing any information loss. Naturally, we could decide to tolerate more information loss (e.g., 1% of samples) and at the same time obtain a higher time savings, and vice versa. However, we believe the above parameter values provide a reasonable savings-to-loss trade off for an operational setting. We therefore use the above values in the following experiments.

4.3.2 *Longitudinal Train-Test Experiments*

To evaluate the effectiveness of our MAXS framework over domain name query events, we performed longitudinal train-test experiments across all malware collection days available for datasets M_A and M_B. Specifically, we proceeded as follows. For dataset M_A, which includes three months of malware analysis data (July, August, and December 2013), we use malware-generated network traces collected during three contiguous days for building the family behavior profiles. Then, we use the next day of malware samples for assessing the time savings and information loss generated by MAXS. This is repeated for every available four-day-long window of malware samples in the dataset. We apply a similar evaluation over M_B (six days worth of malware from November 2014), with the only difference that we only use one day worth of malware samples for training and one day for testing, and therefore we slide this two-day-long window over all available malware collection days.

The results of this set of experiments are reported in Figure 3. As we can see, the time savings are consistently between 30% and 55%, with a median time savings of 42.2% for dataset M_A (the three months in 2013). At the same time, the median domain-based information loss was only 0.25% overall, with a median of only 0.07% samples that caused any information loss at all. Said another way, only a tiny fraction of malware samples are responsible for the 0.25% of domain names that we would miss to observe. It is worth noting also that a 42.2% time savings is substantial, because in essence it would allow us to significantly increase the processing capacity (e.g., malware samples executed per day) for a given bare-metal malware analysis environment.

Similarly, the experiments over dataset M_B (Nov. 2014) produced a 45.5% median time savings, with 0.08% median information loss and only 0.03% samples responsible this loss.

Table 2 additionally reports the overall number of samples that are assigned to a malware family (see Section 3.2), the average execution time before a sample is assigned to a family, and the average execution time before the sample analysis is stopped. As we can see, more than 50% are attributed to a previously learned malware behavior profile, and their execution is preemptively stopped after less than 70 seconds. This is in contrast to blindly executing each sample for the same maximum amount of time (240s and 360s, respectively, for the two malware analysis systems we used), and demonstrates how MAXS is able to achieve significant time savings.

4.3.3 *Evaluating Retraining Interval*

We also performed a number of experiments to evaluate for how long it takes for family behavior profiles learned over a given malware subset to "degrade" and be less representative of new incoming samples. For this, we used the M_A dataset, since it contains samples collected across 77 different days. As for the previous longitudinal experiments, we built numerous training sets, each using three consecutive days of malware data. We randomly selected 20 of such training sets. For each one of these sets, we used the family profiles learned from it, and ran MAXS on the malware collected on the following 10 days. In other words, we use 20 time windows of 13 days each. The first 3 days are used for learning the family profiles, and then these profiles are fixed and applied for making decisions on if/when to stop the execution of the malware samples to be analyzed in the following 10 days.

The median time savings, domain loss, and samples with domains loss for these experiments are presented in Figure 4. Not surprisingly, as time passes (i.e., the farther the test day from the training period) time savings decrease and information loss increases, due to the degradation of the family behavior profiles. Nonetheless, the performance degradation is not dramatic. Even after 10 days after training, we still obtained more than 38% time savings, with less than 0.4% domain loss and less than 0.18% of test samples responsible for that loss. This shows that it is not needed to rebuild the malware family behavior profiles every day to maintain a good level of performance.

4.4 Experiment 2: Leveraging Static Analysis Information

Static analysis based malware "filters" have been studied for example in [12, 21, 24]. In this paper, we do not propose to use new static analysis features to identify potential duplicate samples. Rather, we aim to show that MAXS is generic and can also be applied to cases where an *event* is represented by information derived via static analysis. Our goal again is to save execution time while minimizing information loss. First, given a training set of malware samples, we cluster the samples based on static analysis features. Then, given a new sample, we aim to compare its static analysis features to the family behavior profiles, and decide if we should start executing the sample or not. We make this determination using our MAXS probabilistic decision framework.

To this end, given a malware sample m, we define a "virtual" event e_0 that consists of a vector of static analysis fea-

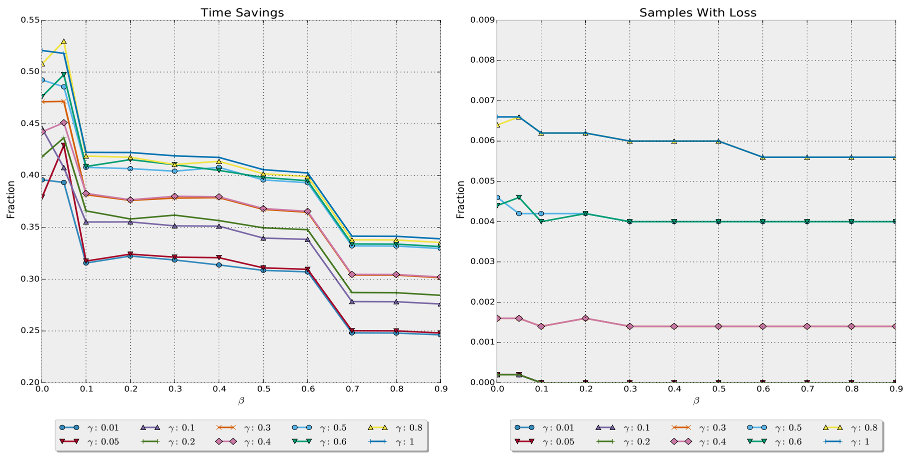


Figure 2: Parameter Selection Experiment

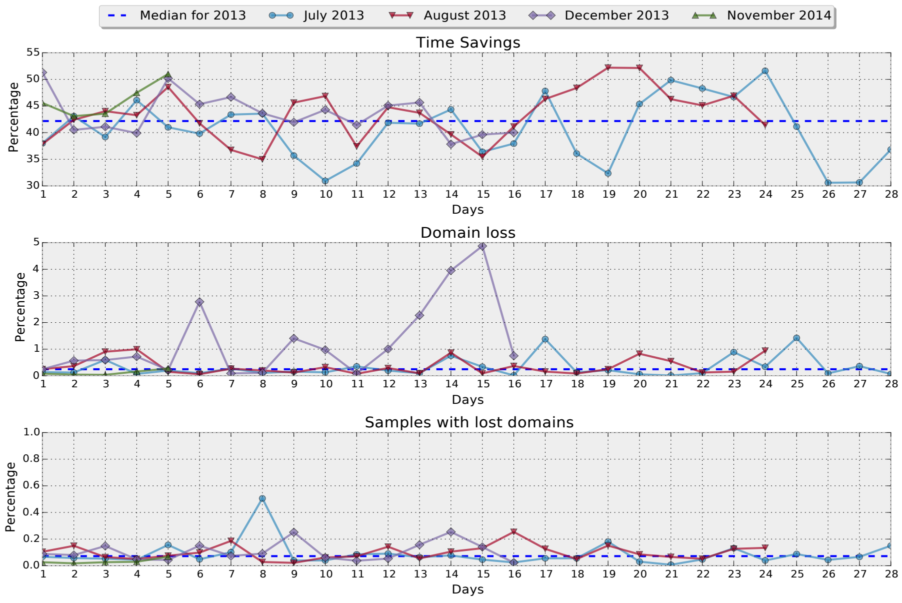


Figure 3: Longitudinal Study Experiment

Dataset	samples	samples assigned to a family	avg. family assignment time	avg. stop time	median time savings
M_A	15,431	9,201	24.4s	69.6s	42.2%
M_B	62,063	34,305	28.3s	50.4s	45.5%

Table 2: Summary of results for longitudinal study experiments.

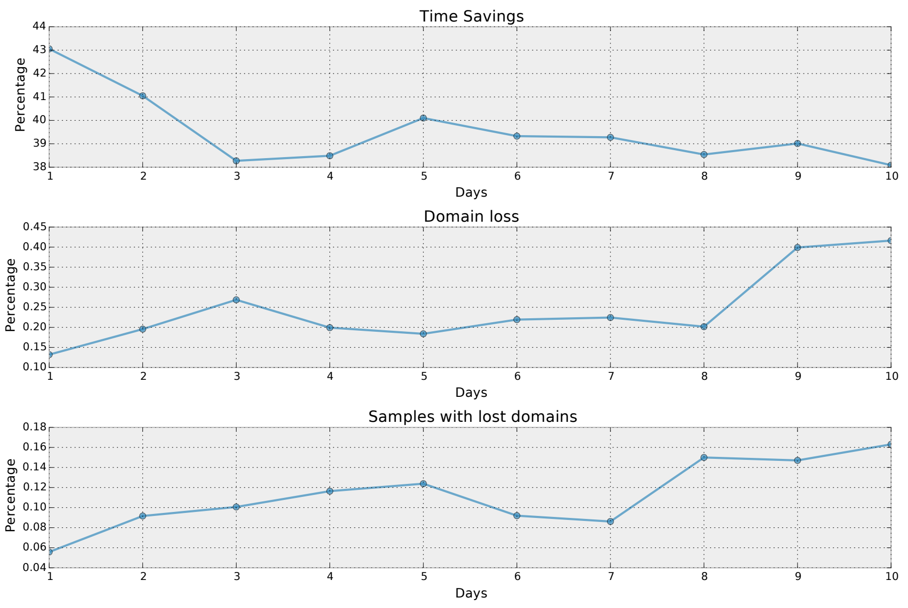


Figure 4: Retraining interval experiments.

tures measured from m. For example, we measure features from PE file headers, such as the number of memory sections, the size of each section, their byte entropy (truncated to a fixed precision limit) etc., in a way similar to [21]. We first use this information to learn malware behavior profiles from a training dataset. In this application, our clustering algorithm is intentionally very simple. We use a binary similarity function, whereby if the static analysis features match perfectly the similarity is equal to one, otherwise we assign similarity zero. Therefore, we put together in the same cluster all samples with identical static analysis features, and separate them from other samples that do not share their features. A family behavior profile is represented by just one (virtual) event e_0 that represents the static features that are shared by all samples in that family.

Now, let m be a new malware sample. Given its own event e_0, we apply MAXS as detailed in Section 3. Notice that the only difference here is that each malware sample "generates" only one event. However, this is not a problem, since our probabilistic decision framework naturally supports this special case. Ultimately, if m is assigned to a previously learned family profile (i.e., its static features match those of a learned malware family), MAXS prevents m from being executed at all. Otherwise, it lets m run for the maximum allowed execution time. If we consider a single sample, the time savings would be either zero or equal to the maximum execution time.

The time savings and domain name information loss can be measured in exactly the same way as described earlier in Section 4.2. For these static analysis based experiments, via pilot testing we set the parameter $\gamma = 0.5$. All other parameters remain the same. The results of these experiments are reported in the first row of Table 3. As we can see, by applying MAXS while using only static analysis information, we can reach more than 37% time savings with 0.22% domain loss.

4.5 Combining Static and Dynamic Analysis

We also performed experiments to show how the static analysis based information and network activities can be "combined" and used by MAXS, using dataset M_B. Specifically, we used all samples that produce any network activity, irrespective of whether or not they make any DNS queries.

We first apply MAXS using static analysis information, as described in Section 4.4. Then, for every malware sample that is allowed to run in this first step, we apply MAXS (sequentially) over the network events generated as the sample is executed, in a way analogous to what described in Section 4.3. Table 3 summarizes the results. The first row reports the combined effect of using both static analysis and network traffic information, whereas the next two rows show the break down of the contributions of the first (i.e., MAXS applied over static analysis information) and second step (i.e., MAXS applied on network events generated by samples that are run after the first step).

The last row shows the results obtained with MAXS by using only network events. As can be seen, there is a significant difference in time savings between the '+Network' and 'Network' rows. This is due to the fact that several malware samples that exhibit similarities in their network behavior also tend to exhibit structural similarity. Hence, such samples would be filtered out at the static analysis stage. As shown in the first row of Table 3, overall MAXS can achieve an average of more than 50% time savings with a domain loss of only 0.3%.

5. RELATED WORK

Bare metal analysis: Balzarotti et al. [4] analyze how malware may change its behavior in emulated environments. In order to deal with this class of "anti-VM" malware, researchers developed techniques to automate malware analysis on bare-metal systems, including BareBox [15] and BareCloud [16]. Our work complements these systems by providing ways to improving their capacity. For example, using

Information	Time Saving %	Domain Loss %	# Domains Lost
Static+Network	50.93 %	0.3 %	114
Static	37.16 %	0.22 %	82
+Network	22.01 %	0.08 %	32
Network	45.5 %	0.08 %	35

Table 3: Average results for a "cascade" decision process using both static analysis information and network events.

MAXS, these systems can double the average number of malware samples they can analyze per day.

Malware clustering: Bayer et al. [18] proposed a system for clustering malware based on system and network information obtained from malware dynamic analysis. Jang et al. [14] used feature hashing to increase the performance of such clustering systems, whereas Perdisci et al. [22] used network level information to cluster HTTP-based malware and generate URL signatures. Wicherski [24] uses features from static analysis to cluster malware.

These works focus on clustering malware, for example as a way to reduce the effort required on behalf of a malware analyst. The above mentioned clustering systems could be used in MAXS to aid the generation of family behavior profiles. At the same time, MAXS is very different from these works. MAXS is a probabilistic decision framework whose main goal is to reduce malware execution time by using sequential hypothesis testing.

Efficient malware analysis: Bayer et al. [5] developed a system to improve the efficiency of dynamic malware analysis systems. While their goals are similar to ours, MAXS differs in a number of important ways. For example, [5] empirically derives a *fixed stop point* when malware could be stopped from running. Unlike [5], MAXS is a generic probabilistic decision framework that uses sequential hypothesis testing to *dynamically* decide if/when to stop executing a malware sample. In addition, MAXS takes information loss explicitly into account, to optimize the trade-off between time savings and missing to observe potentially useful malware intelligence.

SQUEEZE [20] discusses a system that uses multi-path exploration to increase the coverage of C&C domains that are collected by malware analysis. This system is entirely complementary to MAXS, and the two could be used together to respectively increase the efficacy and efficiency of malware execution environments.

FORECAST [19] uses supervised learning techniques over features obtained from static analysis to increase the information obtained from malware analysis. While static analysis component used in MAXS could be improved using an approach similar to FORECAST, it is important to notice that MAXS is a *generic framework* that uses features obtained from both static and dynamic analysis, and that the use of dynamic analysis information significantly improves malware execution efficiency.

6. DISCUSSION AND FUTURE WORK

As most other systems that aid malware analysis, our MAXS framework suffers from some limitations. For example, because MAXS is designed to run "on top" of existing malware analysis environments, it directly inherits some of their issues. Specifically, a given malware variant may decide to sleep for some time (in the order of a few minutes), before performing meaningful network activities. This mechanism, called "stalling" [17], is difficult to detect and circumvent. This is especially true in bare-metal execution environments (e.g., [15]) where system modifications are avoided in an attempt to prevent the malware from detecting the presence of the analysis environment itself. When a "stalling" malware sample is analyzed, MAXS would simply run the sample for the maximum allowed time, thus falling back to the default execution configuration of the underlying analysis system.

A malware author may attempt to evade MAXS, for example by "prepending" the first few network events of a known malware family to a piece of new malware. Thus, MAXS may decide that the new sample belongs to an already seen family, and prematurely stop execution. However, this is counter-productive with respect to the malware author's main purpose, because the "prepended" network events may make the new malware more easily detectable by existing behavioral models or signatures.

In this paper, we focused on demonstrating how MAXS can be applied to static analysis information and domain name queries. In our future work, we plan to apply MAXS to heterogeneous types of network events, including HTTP requests, SMTP traffic, network flow information, etc., besides DNS traffic.

7. CONCLUSION

In this paper, we proposed MAXS, a novel probabilistic multi-hypothesis testing framework for scaling execution in malware analysis environments, including bare-metal execution environments. Our main goal is to automatically recognize whether a malware sample that is undergoing dynamic analysis has likely been seen before, and determine if we could therefore stop its execution early while avoiding loss of valuable malware intelligence.

We tested our prototype implementation of MAXS over two large collections of malware execution traces obtained from two distinct production-level analysis environments. Our experimental results show that using MAXS we are able to reduce malware execution time in average by up to 50%, with less than 0.3% information loss. This shows that, MAXS could be used to significantly cut the cost of bare-metal analysis environments by reducing the hardware resources needed to analyze a predetermined daily number of new malware samples.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation, under grant No. CNS-1149051, and the Department of Homeland Security, under contract No. N66001-12-C-0133. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Homeland Security.

8. REFERENCES

[1] Anubis. https://anubis.iseclab.org/.

[2] Cuckoo sandbox. http://www.cuckoosandbox.org/.

[3] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a dynamic reputation system for dns. In *19th USENIX Conference on Security*, USENIX Security'10.

[4] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[5] U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing.*

[6] L. Bilge, S. Sen, D. Balzarotti, E. Kirda, and C. Kruegel. Exposure: A passive dns analysis service to detect and report malicious domains. *ACM Trans. Inf. Syst. Secur.*, 16(4), Apr. 2014.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08.

[9] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2), 2008.

[10] M. Ester, H. peter Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[11] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08.

[12] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *ACM Conference on Computer and Communications Security*, CCS '09.

[13] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12.

[14] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *18th ACM Conference on Computer and Communications Security*, CCS '11.

[15] D. Kirat, G. Vigna, and C. Kruegel. BareBox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11.

[16] D. Kirat, G. Vigna, and C. Kruegel. BareCloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14).*

[17] C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011.*

[18] C. Kruegel, E. Kirda, P. M. Comparetti, U. Bayer, and C. Hlauschek. Scalable, behavior-based malware clustering. In *16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, NDSS '09.

[19] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. FORECAST: skimming off the malware cream. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011.*

[20] M. Neugschwandtner, P. M. Comparetti, and C. Platzer. Detecting malware's failover c&c strategies with squeeze. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011.*

[21] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recogn. Lett.*, 29(14), Oct. 2008.

[22] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10.

[23] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06.

[24] G. Wicherski. pehash: A novel approach to fast malware clustering. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '09.*