xOSSIG: Leveraging OS Diversity to Automatically Extract Malware Code Signatures

Michael Brengel, Christian Rossow
CISPA Helmholtz Center for Information Security

***Abstract*—We present an automated approach to extract code signatures that serve as the forensic fingerprint of a given malware program. Our high-level idea is to compare the memory contents of a sandbox *before* and *after* infection by a malware. To pinpoint the actual memory changes caused by the malware, and ignore all others, we use a novel concept called *Cross OS Execution*. That is, we execute a malware program on multiple *different but compatible* operating systems (OSes) to identify its memory commonalities, while neglecting OS-specific noise. The commonalities of the dumps therefore contain patterns whose presence is the consequence of executing the malware, i.e., the forensic fingerprint of the malware. We show that we can use two different versions Windows to accurately extract fingerprints of all 17 popular Windows malware families in our test set. These signatures serve to re-identify malware infections in memory dumps with a TPR of 93% and an FPR of 0.15%.**

I. Introduction

Due to packing, polymorphism and malware updates, the security industry faces hundreds of thousands of new malware *samples* daily. Luckily, this huge number of samples can usually be broken down into a small number of malware families, i.e., programs stemming from the same code base. To classify malware samples, analysts rely on *signatures* that map yet unknown programs to the underlying malware family. Such signatures are characteristic byte patterns that express code and/or data that is specific to a certain malware family.

Malware signatures are at the core of the daily operation of malware analysts and serve multiple fundamental activities. First, signatures are heavily used by the AV industry to recognize and mitigate malware. While AV engines augment signatures with behavioral analytics, signatures remain one of the core vehicles to detect *known* threats. Second, threat analysts want to "hunt" files according to a given signature in large malware corpora. For example, an analyst can find all malicious programs of a certain APT campaign by searching for samples that match a signature. Third, leveraging an exhaustive list of signatures of known malware families, analysts can easily spot unknown threats that show anomalous behavior yet do not trigger signatures. Thus, malware signatures are an inherently important asset in the security industry, even beyond AV vendors.

An inherent challenge of malware signatures is that threat actors aim to evade them by using obfusucation techniques such as packing for example. This significantly raises the complexity and amount of manual work required to *create* signatures. The core challenge is that packed malware exposes its full code only during runtime, and hence, any prior offline analysis on packed malware is doomed to fail. Furthermore, even when unpacked, it remains unclear which parts of the exposed patterns are actually *characteristic* for a malware family. Consequently, to create malware signatures, analysts have to (i) manually unpack malware, and (ii) use expert knowledge to search for family-specific byte patterns in the disassembled program.

In this paper, we propose xOSSIG, an automated methodology to extract code signatures for a malware family. Automated signature generation has already proven useful in other contexts, such as in detecting DKOM attacks [1], JavaScript-based exploit kits [2] or rootkits [3]—our focus is malware re-identification, a particularly demanding domain due to packing and polymorphism. We use memory forensics to extract a malware fingerprint that can re-identify a malware family. This comes with several challenges, ranging from the large quantity of unstructured memory content in memory dumps, to malware that tries to hide its presence via obfuscation [4]. To tackle these problems, we execute the malware in a *sandbox* and compare the memory contents *prior to* and *after* infection. Naively, this difference would reveal memory footprints of any process running on the infected system, including benign background processes, whereas we are only interested in the malware footprint. We therefore propose *Cross OS Execution (XOS)* to eliminate such noise from signatures. That is, we execute the same malware in two sandboxes running *different* OSes and intersect both memory footprints to generate signatures.

The derived signatures can then be used to classify malware in a memory-forensic setting. In particular, we can use the derived patterns to find out—just using memory dumps—which malware has infected a system. This replaces the tedious manual investigations of forensic teams trying to identify the malware that compromised a system. That is, analysts do no longer need to (i) inspect running system processes, (ii) identify potentially malicious processes, and (iii) investigate the type of malware by reverse engineering. In addition, our code signatures can classify malware samples in general, simply by executing them in sandboxes and inspecting the memory snapshot after infection. Consequently, although we operate on memory dumps, our approach is fairly generic in terms of malware classification.

Our evaluation with 17 prominent malware families shows the effectiveness of our approach. Given a set of malware samples for each family, we can identify infections caused by other samples of the same family with a true-positive

rate of 93% and a small false-positive rate of 0.15%. We have implemented XOSSIG as an open-source prototype and will release its Python source code, the generated signatures, and the underlying data sets (samples, memory dumps) upon publication of this work.

II. MOTIVATION

Overall Goal We aim to generate code signatures that, once derived, can identify which malware family has infiltrated a system M. Whereas individual malware executables of a family can be short-lived due to hard-coded data such as C2 server addresses or updated anti-virus signatures, the malware family continuously operates for many years. This limits the number of active malware families (hundreds) to be significantly smaller than the number of malware samples (hundreds of thousands *per day*).

Forensic Setting Our approach operates in a purely *memory-forensic* setting where we assume a memory dump D_M of an infected system M. We then detect if D_M contains patterns that reveal the presence of a known malware family. Prior academic malware classification attempts have relied on either perfect static analysis or on rich runtime information such as system call traces to classify malware [5], [6], [7], [8]. However, while such rich information is available in malware sandboxes, live systems cannot use fine-grained monitoring of processes for performance reasons. Once a live system is infected, we thus lack methodologies for classifying the malware behind the infiltration.

Challenges Identifying characteristic artifacts of malware families is one of the daily routines of malware analysts. An analyst would manually analyze malware to locate portions of code and/or data which seem characteristic enough and create a signature based on this analysis. There are, however, two problems with this approach compared our methodology. First, it requires time-consuming manual effort and domain-specific knowledge, which is in stark contrast to our goal of automation. Second, which malware patterns are sufficiently characteristic depends on the perception of the analyst. Any such subjective rule extraction conflicts our goal to rigorously define and systematically determine characteristic memory patterns.

Straw-Man Proposal Extracting forensic malware fingerprints is non-trivial, as the following first idea draft demonstrates. To expose the characteristic patterns of some malware sample s, we can infect a sandbox M with s, wait for some time, and take and analyze the memory dump D_M. This complete memory snapshot includes non-malware related artifacts of benign processes, libraries, services and so on, whereas we are purely interested in extracting malware-specific patterns alone. For example, a memory dump taken of a machine running a modern version of Windows can be multiple GiB large, whereas the interesting portion is an almost negligibly small subset with a size in the order of KiB.

The core challenge is extracting the characteristic malware patterns from the memory dump in an automated way. An immediate first idea that comes to mind compares the memory content of the sandbox *before* and *after* infection. By inspecting which memory contents have changed during infection, we can extract the malware footprint. However, we will show that this naïve idea does not perform well for multiple reasons. First, not all memory changes are caused by the malware, as benign background processes continue execution and change memory in parallel to the malware. Second, even when inspecting malware-related changes only, the contents may stem from libraries or boilerplate code used by the malware.

Cross-OS Execution We tackle these aforementioned challenges and extract malware fingerprints with a novel concept which we call *Cross-OS Execution*. On a very high level, the idea of Cross-OS Execution is to execute a malware sample s on *two* systems M_1 and M_2 running *different but compatible* OSes, and then to compute the fingerprint of s by identifying the forensic commonalities of D_{M_1} and D_{M_2}. The fingerprint serves to identify an infection with the same family in an arbitrary memory dump, following the intuition that samples of the same family share the same forensic characteristics. The term "different but compatible" OSes here means that the malware will execute on both systems, but the OSes are different in the sense that D_{M_1} and D_{M_2} do not share most memory content other than the portions belonging to s. Consequently, we implicitly abstract from any memory changes caused by background activities in the OSes. In this work, we will focus on the Windows OS due to its backwards-compatible nature and the fact that many different versions of Windows are in use [9].

III. METHODOLOGY

Figure 1 depicts our overall process to extract the forensic fingerprint for a sample s including all the intermediate steps. First, we execute s on two sandbox systems M_1 and M_2 to obtain their post-infection memory snapshots. Second, to abstract from boilerplate memory contents, we search for memory portions that are unique within a memory dump. Third, we identify memory contents that are common between the two cross-OS executions. Fourth, we use pre-infection information to remove memory contents that were already present *before* infection, or would have happened also without infection. Fifth, as we aim for code signatures, we extract all code pages from the remaining unique common memory contents, and discard non-code fragments. Finally, we use this code to extract the forensic fingerprint—a set of n-grams N_s—for s. In the following, we will describe these steps in detail.

A. Memory Dump Representation

In the first step, we execute s on the two machines M_1 and M_2 for a predefined amount of time after which we take the memory dumps D_{M_1} and D_{M_2} of both machines. Our assumption is that the malware executes on both systems, while the OSes offer code and data diversity to reduce the amount of memory artifacts they share. In our experiments we have chosen Windows XP and Windows 7. Comparing memory dumps in their entirety is not productive, so we

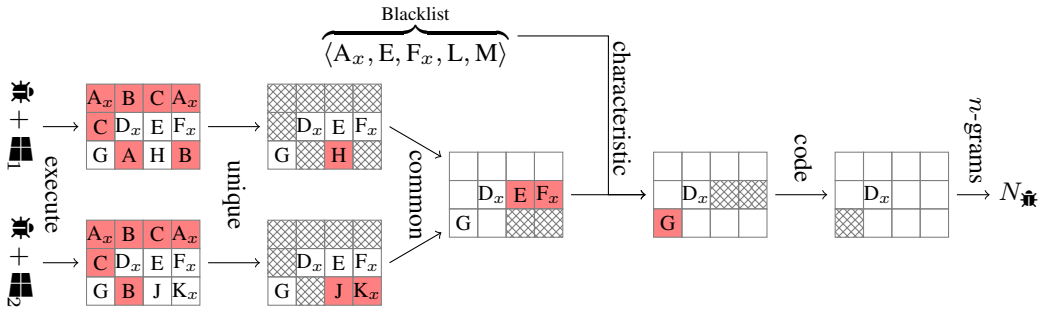

Fig. 1. A high-level simplified overview of our approach. We start with a sample that gets executed in two machines running different but compatible versions of Windows. This yields two memory dumps consisting of a sequence of memory pages. We then compute the unique common pages of both memory dumps, apply a noise reduction and a code heuristic after which we compute the n-grams of the final pages to compute the forensic fingerprint N. Pages with the x subscript are executable.

break them down into smaller units that follow common OS semantics. That is, we view a memory dump D as a set of equally sized *pages*, i.e., $D = \{P_1, \ldots, P_k\}$, which reflects the concept of *virtual memory* of modern OSes. Our intuition is that if malware exposes itself on a system, it will do so using dedicated memory pages that store malware code or data.

B. Unique Common Page Extraction

Our main objective is to find pages that are shared between the dumps D_{M_1} and D_{M_2}. A simple solution to this problem would be to intersect the pages of both dumps. However, the memory pages of the malware might not necessarily be exactly byte-wise equal in both memory dumps. For instance, if the malware uses relocations, it will change offsets in the code pages and thus will be slightly different in both memory dumps. It is also not uncommon for malware to use data and code on the same memory page, and data is more likely to be modified in a non-deterministic fashion. To address this problem, we will instead search for common *chunks*, i.e., non-overlapping equally-sized segments $\langle C_{i,1}, \ldots, C_{i,m}$ of a page $P_i\rangle$. We will compute the common chunks of both dumps and use them later to get the pages containing them.

To compute the common chunks we could compute and intersect the chunks of both dumps. However, this is again too simplistic for one main reason. In order to cope with the previously mentioned problems, the chunk size needs to be substantially smaller than the page size. Due to this fact, a common chunk might contain non-characteristic patterns such as byte paddings emitted by compilers or boilerplate code snippets. Given that we want to get the pages containing the common chunks, we would get all pages of the dump that contain these non-characteristic boilerplate patterns, which is undesirable. Instead, we restrict ourselves to *unique* chunks—chunks that appear only once in a memory dump—to ensure that we will only consider characteristic forensic artifacts.

However, we cannot require such *strict* uniqueness. Because code pages are usually mapped as Copy-on-Write pages, a modification of such a page will make a copy of the page in physical memory, apply the modifications to the modified page, and remap the virtual page to this copy of the page. If the page gets modified (e.g., by a relocation), chunks of modified pages which are *not* in the modified area are not strictly unique anymore—they will be present both in the modified page and the original page. We thus relax the definition and not only call a chunk unique if it occurs only once in the memory dump, but also if it occurs at *the same page offset* in multiple "similar" pages according to a distance metric δ and a threshold ϵ.

Coming back to our original objective, i.e., finding common pages of both memory dumps, we need to go back from chunks to pages. To this end, we first find the *unique common chunks* $\mathcal{C} = \mathcal{C}_1 \cap \mathcal{C}_2$ of both dumps. What is left to do is abstracting from $\mathcal{C}$ to *similar unique pages*, complicated by the fact that a unique common chunk may be part of *multiple* pages. To compute the set of similar unique pages $\mathcal{P}_i$ for i in $\{1, 2\}$, we therefore randomly choose exactly one page of D_{M_i} containing a chunk $C \in \mathcal{C}$ and add this page to $\mathcal{P}_i$. This whole algorithm has linear complexity, as we only need to go through all the pages once, compute the chunks and keep track of the relationship between pages and chunks and the positions at which the chunks occur.

To ensure that the fingerprint is based only on true commonalities, i.e., similar unique pages that occur in both memory dumps, we need to consider only pages that are sufficiently similar. We know that pages in $\mathcal{P}_1$ share *at least one* characteristic chunk with pages in $\mathcal{P}_2$. We thus compute the intersection I, i.e., the set of *unique common pages* that are sufficiently similar between $\mathcal{P}_1$ and $\mathcal{P}_2$. The compute I, take every $P_i \in \mathcal{P}_1$ and greedily look for a closest candidate $P_j \in \mathcal{P}_2$ with respect to the Hamming distance δ such that $\delta(P_i, P_j) \leq \epsilon$ is minimal. The Hamming distance δ captures the modifications that we aim to abstract from, i.e., drop-in replacement of bytes as opposed to complex modifications that shift and re-align bytes. We ignore chunks present at multiple different page offsets, as this indicate that the chunk is too generic. Although this naive algorithm has quadratic complexity, as we will show, the sets of characteristic pages are so small that this does not have a noticeable negative impact on performance.

C. Removing Execution-Unrelated Memory Content

The unique common pages could still contain pages which were emitted by the OSes and any background processes. Even

if the two OSes are mostly different with respect to their forensic footprint, some components may survive various OS generations. To reduce such noise, we use a *training phase* in which we run both OSes without any malware executing for the same amount of time that we will execute malware. We then compute the unique chunks of the resulting *idle memory dumps* as described previously. Those unique chunks are then intersected and added to a set B, which we will call the *blacklist*. We repeat this process until the blacklist saturates. In our experiments, after just two idle executions, no new unique chunks were found in follow-up runs, indicating that B converged towards the noise of both OSes. The blacklist consists of $27,721$ chunks—98.26% of which were the result of the first execution. To remove pages containing noise, we discard any page from $\mathcal{P}_1$ and $\mathcal{P}_2$ which contains a blacklisted chunk $C \in B$. We call the sets of pages surviving this step the *characteristic pages*, as they are characteristic for a malware sample s, and refer to them as $\hat{\mathcal{P}}_1$ and $\hat{\mathcal{P}}_2$.

D. Identifying Code Pages

Given the intersection I, we perform a filtering step that aims to identify pages which are likely to consist of *code* (machine instructions). While malware signatures used in practice usually target data such as characteristic strings, we aim at code-based fingerprints instead. Data-based signatures suffer from the fact that an attacker can easily hide them from a forensic point of view with on-demand encryption and decryption. In contrast, code-based signatures are more resistant since they would require an attacker to substantially modify their malware at the code level to ensure that the forensic fingerprint is completely different.

We could identify code pages by examining the page tables and the internal OS data structures to find out if a page is marked as executable. However, our approach should be completely independent of any OS-specific idiosyncrasies. Also, attackers can manipulate those data structures to make it seem that the pages are not executable by making them executable on demand. Furthermore, it is not guaranteed that VMI can access the page tables of terminated processes (e.g., of short-lived malware droppers) to determine these permissions. Instead, to identify x86 code pages, we use a heuristic that disassembles a page using the Capstone disassembler [10] and looks for x86-specific instruction sequences which are likely to be found in code (e.g., `push/push/call` or `call/test/je`). To account for alignment issues, we disassemble each page at 15 consecutive offsets, since that is the maximum length of an x86 instruction. We consider a page $P \in I$ a code page if it contains 10 or more such patterns. All such code pages become the final *intersection of characteristic code pages* I_X that captures malware-specific code pages.

E. Signature Generation

With the intersection of characteristic code pages I_X, we can compute the fingerprint of N_s, which ca be used as signature to identify other samples of s's family in memory dumps. Simply looking for the whole pages of I_X in other memory dumps is unlikely to be successful. Instead, we compute the n-grams of the pages of I_X. This step follows the intuition that even if the malware author applies many modifications to the code of the malware, it is likely that at least some byte sequences remain unchanged to successfully re-identify the family. We calculate the n-grams of the malware sample s, denoted by N_s, by moving a sliding window over each page $P \in I_X$. We only add an n-gram if its covered chunks are all uniuque in the memory dump that the page originated from to ensure the quality of the signature. Additionally, we ignore an n-gram if it is part of the blacklist B.

The output of this final step is a fingerprint N_s, i.e., the set of n-grams of the intersection of characteristic code pages. Our intuition is that N_s can be used to identify different samples belonging to the same family as s by searching for n-grams of N_s in an arbitrary other dump. We determine that a memory dump matches N_s if the number of n-grams shared between the dump and N_s is sufficiently large, which we will assign concrete meaning to in the next section.

IV. Evaluation

We now evaluate our methodology, which includes a brief overview of our setup, the choice of malware families, the execution of samples and the extraction of characteristic n-grams. After that, we outline our parameter selection process to find suitable values for the chunk size, the page similarity threshold ϵ and the size of the n-grams. This is followed by an analysis of the intersecting behavior of the extracted n-grams between and among families. Additionally, we measures true positives and false positives by checking if we can detect malware families in memory dumps to assess the overall effectiveness. Finally, we give insights into the performance of our methodology.

A. Setup and Implementation

In our prototype implementation, we execute and monitoring malware samples using VirtualBox. Our methodology is completely agnostic of the choice of hypervisor, and therefore a stealthier hypervisor such as Ether [11] or a bare-metal sandbox [12] can be used as a drop-in replacement.

We leverage the diversity of the Windows ecosystem by running a malware sample in multiple VMs with different versions of Windows. In particular, we use Windows XP with 512 MiB RAM and Windows 7 with 2 GiB RAM. We run each sample in both VMs for two minutes and record its network traffic and collect a full memory dump.

Creating a ground truth dataset for our evaluation is challenging, as it involves several manual steps. First, we have to manually label samples according to their family, as AV labels are known to be noisy, even if cleaned appropriately [13]. Second, we have to ensure that the malware samples become active in our evaluation, i.e., they expose their actual behavior and memory footprint during execution. Instead of using a large low-quality dataset, we create a vetted and reasonably-sized high-quality dataset. To be representative, we include families that belong to all prevalent threat types, such as

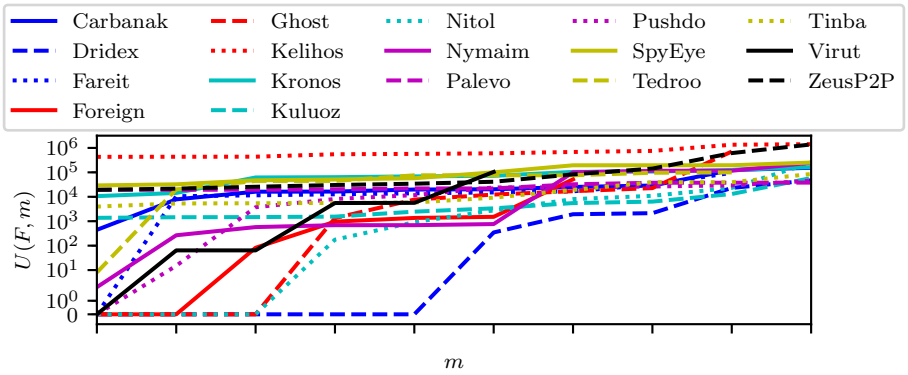

Fig. 2. The uniformity score $U(F, m)$.

banking trojans, APTs/RATs, spam bots and downloaders. Our malware dataset consists of 197 samples from 17 popular malware families[1]. For each of these families, we use (approximately) equally-sized groups to avoid biases of otherwise-overpopulated families. After executing each sample, we manually verified that the sample became active by inspecting the network traffic of both VMs and by ensuring we found the family-specific C2 activity.

B. Parameter Selection

Our methodology uses three parameters that we need to find suitable values for. Those are the chunk size, the page similarity threshold ϵ and the size of the n-grams. To asses the quality of the chosen parameters, we need to give a metric for the quality of an n-gram set N_s. To do so, we compute for each sample how well the n-grams intersect with the n-grams of its own family and how they intersect with other families. Formally, let F_s be the malware family of a sample $s \in F_s$. We define N_s to be the set of n-grams computed for s as described in Section III and let $N_F = \bigcup_{s \in F} N_s$ be the n-grams of a family F. For each sample s, we now compute the sanity score $S(s)$, which is the fraction of n-grams that intersect with the family of s but not with any other family, i.e., $S(s) = |(N_s \cap \bigcup_{\substack{s' \in F_s \\ s' \neq s}} N_{s'}) \setminus \bigcup_{F \neq F_s} N_F| / |N_s|$. Note that we remove all n-grams of other families for calculating $S(s)$. This is important because samples of different families might share n-grams due to the usage of shared libraries or well-known packers, for example. Similarly, we compute the complementary sanity score $\hat{S}(s)$, which describes the fraction of n-grams that do intersect with families other than F_s, i.e., $\hat{S}(s) = |(N_s \cap \bigcup_{F \neq F_s} N_F| / |N_s|$. To find the best parameters, we calculate an F-score-like measure $f(s) = \frac{2S(s) \cdot (1 - \hat{S}(s))}{S(s) + (1 - \hat{S}(s))}$ for each sample and different configurations of chunk size, ϵ and n and choose the configuration which yields the largest value for f on average. We let possible candidates for the chunk size range over $4, 8, 16, \ldots, 4096$ and possible candidates for ϵ over $16, 32, 64, \ldots, 512$. We determine that n should be equal to the chunk size as we consider both the characteristic length of a string of bytes that uniquely identifies a malware. This gives us a total of 66 configurations to test. Our findings indicate that the best configuration is 32 for the chunk size and n, and 128 for the page similarity threshold ϵ.

[1]Carbanak, Dridex, Fareit, Foreign, Ghost, Kelihos, Kronos, Kuluoz, Nitol, Nymaim, Palevo, Pushdo, SpyEye, Tedroo, Tinba, Virut, and ZeusP2P

C. Intra- and Inter-Family Analysis

For the remainder of this evaluation, to avoid imbalances caused by more prevalent families, we will work on a subset of (up to) 10 samples per family. We chose these ten samples such that they are evenly distributed over time (their age). It is important to understand how similar the sets of n-grams of a family are among themselves. The previously computed sanity score S can be very high for a sample s if there is just one sample $s' \in F_s$ of the same family such that they both share a significant number of n-grams. This means that S ignores how many samples share how many n-grams, which means that S is not a good measure for understanding the uniformity of the n-grams. Instead, we calculate the uniformity score of a family F, given by: $U(F, m) = \max_{\substack{F' \subseteq F \\ |F'| = |F| - m}} \left\{ \left| \bigcap_{s \in F'} N_s \setminus \bigcup_{F'' \neq F} N_{F''} \right| \right\}$. To put it simply, $U(F, m)$ is the largest number of n-grams that are shared among a subset of F and no other family after removing m samples from the family. In other words, the larger $U(F, m)$ is for small m, the more uniform the n-grams of F are, which gives us an idea of the effectiveness of our approach. In Figure 2 we can see the uniformity scores for all families and all possible values of m. For 10 out of 17 families the uniformity score is above zero for $m = 0$, indicating that several n-grams are shared among all samples for those families. For the remaining 7 families (Dridex, Fareit, Foreign, Ghost, Nitol, Pushdo and Virut), the uniformity score is equal to 0 for $m = 0$ and increases for larger m. However, in all cases it is shown that having enough samples of a family yields a characteristic set of n-grams for a family.

To investigate the reasons for the cases with low uniformity, we manually analyzed the corresponding memory dumps. In the cases of Dridex, Fareit, Foreign, Pushdo and one sample of

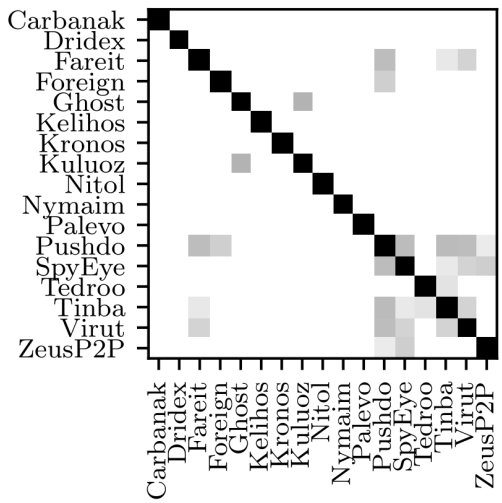

Fig. 3. log heatmap for n-gram similarity.

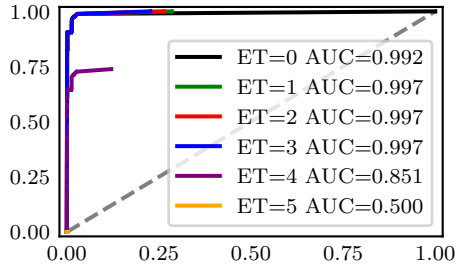

Fig. 4. ROC curves of our memory dump classification for varying entropy thresholds.

Nitol, the problem was related to the Cross OS Execution. We discovered that the intersection of characteristic code pages I_X did not contain the entire actual code of the malware for some samples. This is because the memory dumps contained the code of the malware multiple times at differently aligned locations. Because of that, we failed to identify the unique chunks, since we require that unique chunks always appear at the same offsets. We will further discuss this phenomenon in Section V. For Ghost and for 1 Nitol sample, we discovered that the corresponding processes indeed did not share n-grams with other processes of malware samples of the same family. We did not perform this experiment in the case of Virut, as it injected itself into other processes and analyzing the malware executable was therefore not easily possible.

These results indicate that some families are more uniform with respect to their code. Families like Tinba, Palevo or ZeusP2P have uniformity scores that are high for small m and remain relatively stable for larger m. In contrast, other families like the 7 listed above have scores as low as 0 for small m. Other families like Carbanak start out with a uniformity score of 444 for $m = 0$ but are less constant, with its uniformity score increasing by two orders of magnitude for $m = 2$. While a uniformity score of 444 might sound small at first glance compared to other scores of more uniform families, it is worth noting that this represents 444 n-grams, which only occur in the intersection of all Carbanak samples and no intersection of any other family.

To assess how suitable the n-grams are for classification, we use the overlap coefficient to compare families: $\hat{J}(F_1, F_2) = \frac{|N_{F_1} \cap N_{F_2}|}{\min(|N_{F_1}|, |N_{F_2}|)}$. The heatmap in Figure 3 plots the overlap coefficient for all families. Although the overall number of shared n-grams is rather small (note the log scale), we can see an overlap for some families. For example, Kuluoz and Ghost share $6,429$ n-grams. Such cases can be explained by the usage of similar third party libraries such as a third party cryptography library or a packer for example.

D. TP/FP Analysis on Memory Dumps

We now evaluate how our approach can be used to detect a malware family in infected memory dumps. To do this, we partition each set of samples for each family F_i in two equally-sized and disjoint subsets F_{i_1} and F_{i_2}. The partition is done such that the samples in F_{i_2} are newer than the samples in F_{i_1}. The rationale behind this is that we want to test whether we can detect newer samples in F_{i_2} by learning the fingerprint of the older samples in F_{i_1}. For each family F_i we compute the n-grams of the samples of F_{i_1}, i.e., $N_{F_{i_1}}$. Then, we intersect the n-grams of all memory dumps of the samples of F_{i_2} with those n-grams.

The detailed results of this experiment are depicted in Table I. Each cell displays the average fraction of the n-gram set contained in a memory dump as well as the average Shannon entropy of those n-grams per family. To improve readability, we omit cell numbers if the average fraction of matching n-grams is smaller than 0.01%. We observed that the n-grams for each family have the largest intersection with their own family, which also yields the largest entropy on average. In the cases where there was at least some intersection between the n-grams and the memory dumps of distinct families, we can observe that the intersection is smaller and also that the entropy is low. For example, 29.33% of the n-grams of SpyEye match on average with the SpyEye memory dumps with an average entropy of 4.28. If we compare this with the ZeusP2P memory dumps, only 0.01% of the n-grams of SpyEye match on average with a lower average entropy of 1.45.

We can now try to find a good compromise between false positives and false negatives by setting appropriate thresholds that determine if a memory dump matches a family. In our context, a false positive arises if we falsely flag a memory dump as belonging to a certain family. Similarly, a false negative occurs if a memory dump contains a malware process of a known family, but we fail to flag the memory dump as such. To determine where an n-gram set matches a memory dump, we set a threshold for the fraction of n-grams of the n-gram set which are contained in the memory dump as well as a threshold for the average entropy of those matching n-grams.

	Carbanak	Dridex	Fareit	Foreign	Ghost	Kelihos	Kronos	Kuluoz	TPR	FPR
Carbanak	5.51% 4.26	-	-	-	-	-	-	-	5/5	0/77
Dridex	-	8.96% 4.07	-	-	-	-	-	-	5/5	0/77
Fareit	-	-	7.34% 4.01	-	0.02% 1.60	-	-	-	5/5	0/77
Foreign	-	-	-	16.48% 4.12	-	-	-	-	4/4	1/78
Ghost	-	-	-	-	6.54% 4.02	-	-	0.68% 4.19	5/5	0/77
Kelihos	-	-	0.02% 2.47	0.12% 4.10	-	20.07% 4.11	0.06% 3.77	-	5/5	0/77
Kronos	-	-	-	0.35% 4.03	-	0.03% 3.08	18.35% 4.12	-	5/5	0/77
Kuluoz	-	-	-	-	-	-	-	20.14% 4.02	5/5	0/77

	Nitol	Nymaim	Palevo	Pushdo	SpyEye	Tedroo	Tinba	Virut	ZeusP2P	TPR	FPR
Nitol	2.29% 3.86	-	-	-	0.01% 2.81	-	-	-	0.03% 1.54	5/5	0/77
Nymaim	-	14.04% 4.19	-	-	-	-	-	-	-	2/5	0/77
Palevo	-	-	48.93% 3.92	-	-	-	-	-	-	5/5	0/77
Pushdo	0.06% 3.29	0.04% 4.22	-	26.02% 3.94	0.04% 3.82	0.04% 3.67	-	-	0.04% 1.52	5/5	0/77
SpyEye	-	-	-	-	29.33% 4.28	-	-	-	0.01% 2.59	5/5	0/77
Tedroo	-	-	-	0.27% 0.86	-	46.69% 4.17	0.47% 0.85	-	0.01% 1.45	5/5	1/77
Tinba	-	-	-	-	-	-	5.17% 4.03	-	-	5/5	0/77
Virut	-	-	-	-	-	-	-	0.48% 4.18	0.04% 1.54	0/3	0/79
ZeusP2P	-	-	-	0.08% 2.14	0.26% 2.48	-	-	-	22.60% 4.17	5/5	0/77

TABLE I

TP AND FP ANALYSIS. EACH CELL DISPLAYS THE AVERAGE FRACTION OF THE n-GRAM SET CONTAINED IN A MEMORY DUMP AS WELL AS THE AVERAGE SHANNON ENTROPY OF THOSE n-GRAMS PER FAMILY. LOW AVERAGE MATCHING FRACTIONS ($< 0.01\%$) ARE OMITTED FOR READABILITY.

To understand the compromise of different thresholds for those two criteria, consider Figure 4, which depicts the ROC curves for our classifier. We plot 6 different ROC-curves for each entropy threshold in $\{0, \ldots, 5\}$ and increase the threshold for matching n-grams with a step size of 0.001. The fact that the classifier with an entropy threshold of 0 has an AUC of 0.992 indicates that the matching number of n-grams alone is already a good matching criterion. An entropy threshold of 5 is unsurprisingly too restrictive, but our results also show that 4 is not optimal. To identify the optimal thresholds, we optimized the F-score given by $F_\beta = \frac{(1+\beta^2)\cdot \mathrm{TPR}}{(1+\beta^2)\cdot \mathrm{TPR} + \beta^2 \cdot \mathrm{FNR} + \mathrm{FPR}}$ for $\beta = 0.25$ to penalize false positives. This yields 3 for the entropy threshold and 0.18 as a threshold for the fraction of matching n-grams, i.e. 1.8% of n-grams have to match. This results in a true positive rate (TPR) of 93% and in a false positive rate (FPR) of 0.15%, which is also reflected in the *TPR* and *FPR* column of Table I. As expected due to the uniformity experiments in Section IV-C, the two families Virut and Nymaim have FNs—a fact that was already visible during training. Hence, the uniformity analysis allows XOSSIG users to decide if the resulting signatures are sufficiently descriptive for them, notably before *before* risking FNs.

Qualitative FP Analysis There are two families with each a single FP. In the case of Foreign the n-grams match with a Pushdo memory dump. We discovered that this is because of a portion of code that is shared between the samples. The code uses certain magic numbers such as `0x6C6C642E` (".dll" in ASCII) and uses the `fs` register to access the Process Environment Block. This is a strong indication that this is low-level code which does stealth API calls. We presume that this code is part of a third-party module, such as a packer that is used by both families. In the case of Tedroo, the n-grams match a Tinba memory dump. Using manual forensic analysis similar to the previous case, we found out that the code containing the matching n-grams belongs to the Visual C++ runtime library of Microsoft. This problem of sharing libraries will be discussed in Section V.

Optimizing Detection Rate In a real-world deployment of our methodology, one could optimize the detection capabilities on a per-family basis. That is, one could set different thresholds for different families. For example, while 5.17% of the n-grams of Tinba match on average with the Tinba memory dumps, this number differs largely for other families as for example in the case of Tedroo, where 46.69% match on average. Similarly, one could construct the n-gram sets with different configurations for different families. While one family might work well with a chunk size of 32 and an epsilon of 128, this does not have to be the case for other families. Our parameter selection showed that this is the best configuration on average for all families we considered, but we found that the optimal configurations are different among the families. For example, the scores of Nitol are optimal when using a chunk size of 32, but in the case of Tedroo setting the chunk size to 256 yields better results than 32. Although this is a more fine-tuned approach, it would still be a one-time effort as this is only required once per family in the learning phase.

E. Performance Evaluation

We use a non-optimized and single-threaded prototype written in Python to evaluate the performance of our methodology. In the following, we distinguish between the one-time effort

to extract fingerprints for a sample, and a repeating effort of matching a new memory dump against given fingerprints.

Fingerprint Generation We first measured how long the fingerprint extraction took, that is from starting the VMs with each sample s until the generation of the fingerprint N_s. However, we subtracted the two minute execution time of the malware from these measurements as this is an uninteresting factor for our performance evaluation. This process took 4.21 min on average (median 3.78 min, standard deviation 1.28 min). The bottleneck here is the comparing step described in Section III-B, since it uses an algorithm with quadratic complexity, which could become a problem if we have too many pages in the intersection I_X. If we considered this bottleneck a problem, we could optimize this step by only comparing pages that share a unique chunk.

Memory Dump Scanning On average, it took 1.98 min to scan a 512 MiB memory dump for all the 34 n-gram sets (median 1.97 min, stddev 0.06 min). The influencing factor is the size and entropy of the memory dump and performance scales linearly with the number of unique pages in the dump. To evaluate the scalability of our approach, we simulate the presence of more families. We started with the 17 n-gram sets and iteratively doubled the number of sets by randomly creating new sets of the same size as the original ones. We then scan a random memory dump using those sets and tested up to 136 malware families. We could not observe significant deviations apart from common measurement deviations, thanks to the amortized $O(1)$ complexity of our n-gram matching procedure. Our approach, both signature generation and scanning, thus easily scales to larger number of families.

V. Discussion

We will now discuss our assumptions, the limitations of our approach and how future research could tackle these challenges. Furthermore, we provide an in-depth discussion on further real-world use cases of our methodology.

A. Limitations

Our evaluation showed that our methodology copes well with most of the chosen malware families. However, there are a few cases that might require changes to our assumptions.

Split Behavior on Different Operating Systems We exploit the fact that malware samples typically expose the same behavior and code on different OSes. While this was true for all samples in our dataset, in principle, malware could use OS-specific payloads. We believe that this problem would primarily not occur for the malware itself, but could be caused by an OS-aware dropper that implants OS-specific malware samples. To some extent, this challenge could be tackled by multipath execution [14] to enforce that both implants are executed, with the clear limitation that malware must not use OS version-specific features. Furthermore, to reduce the risk of OS-specific behavior, one could choose more similar OSes. We conducted an experiment using xOSSig for Tinba, but this time in two Windows XP VMs with different patch levels, SP 2 and SP 3, respectively. Although the construction of the blacklist B as described in Section III required more runs to converge, the resulting n-gram set was almost identical to the one we used for our evaluation.

At the same time, our key idea to execute malware on two different OSes is vital. To illustrate this, we conducted the same experiment on a single OS and patch level (Windows XP SP3). This time, although the blacklist converged again after more runs, but the extracted set of characteristic and executable pages of the Tinba sample was an order of magnitude *larger* than before. This stark increase of supposedly-characteristic code indicates the presence of noise which would render the extraction useless for classification. That is, for same-OS executions, the blacklist B only abstract from noise of the idle state. However, there is deterministic noise upon malware execution that is not captured by the blacklist and thus falsely becomes part of the fingerprint.

Non-Presence in Memory A fundamental assumption we make is that the malware code is present in memory at the time the memory dump is taken. This follows the intuition that persistence is an inherent property of malware, as adversaries have an incentive to ensure that their malware runs all the time. This is true for the vast majority of malware types in the wild, except for some types of malware such as ransomware, which terminates after decrypting the files of the user. This could be solved by a more sophisticated dumping mechanism which takes a dump of the whole memory or certain memory areas after process termination using secure VM monitoring [15].

Complex Packing xOSSig focuses on the malware memory footprint and thus copes with the diverse set of runtime packers used by the prevalent malware families in our dataset. This supports the findings of Ugarte-Pedrero *et al.* who observed that the vast majority of in-the-wild malware maps its entire unpacked code segment in memory [4]. Having said this, malware in principle can deploy more sophisticated packing schemes. For example, the x86 code identification that we use would fail to detect the malware's code if VM-based obfuscation was used. Here, we would have to recognize the bytecode of the custom virtual machine. Similarly, if packers encrypts and decrypts code parts (e.g., functions) on demand, we can only learn the n-grams of the code which is decrypted at the time of taking the memory dump. Adapting xOSSig for these needs, e.g., using a more sophisticated memory dumping procedure is an interesting future work challenge.

Alternative Bytecode and Interpreted Code Our current prototype is tailored to x86 code, given its prevalence in the Windows malware domain. Consequently, the prototype cannot identify .NET bytecode or interpreted code (e.g., Python, PowerShell) such as used by fileless malware. This can be addressed by extending our code identification strategy to .NET bytecode and other bytecode, or to interpreted languages, respectively. This does not fundamentally change our concept and is mostly an engineering effort that requires to study another ISA and/or interpreter.

Evasion When xOSSig finds wide adoption in practice, malware authors may aim to subvert the fingerprint extraction. In principle, attackers could use xOSSig to find out whether

or not their current code exposes characteristic memory artifacts. There are several strategies to remove characteristic memory contents, which we will briefly discuss. First, malware authors may duplicate their code to subvert our uniqueness assumption of unique chunks. We could solve this problem by relaxing our notion of *unique chunks* by allowing such a chunk to appear at multiple offsets and requiring the pages which share those offsets to be similar. Second, malware could use fine-grained code randomization approaches [16], [17] to remove any invariance from its code such that no single chunk will be shared for multiple executions. However, given code randomization has not found wide adoption in practice as it adds suspicious randomization behavior that would ease detection. Fhird, malware could try to deploy anti disassembly tricks such as overlapping instructions and obscuring control flow, which would fool our code detection technique. This could be solved by using a more complex machine-learning assisted code detection technique [18].

B. Further Use Cases

We believe that our general methodology also allows several use cases other than identifying known malware families in infected memory dumps.

Malware Labeling The increased adoption of behavioral malware detection methodologies has reduced the quality of malware labels. Frequently, labels are too generic to be usable by experts. We envision that our approach could be used as a mechanism to label malware executables. For example, AV vendors or sandbox operators may want to extend their knowledge of malware families and run our method as an add-on to existing classification techniques.

Efficient Storage of Forensic Evidence Our approach helps to persistently store the forensic footprint of a malware. Sandboxes that execute hundreds of thousands of programs per day can only save compact and aggregated information of the malware activity. Persistently storing complete memory dumps is a challenge for many sandbox operators due to the sheer file size. By storing the set of extracted characteristic pages only, we significantly reduce the storage capacities that are required compared to storing full memory dumps. The extracted characteristic pages were orders of magnitude smaller (KiB instead of GiB) than entire memory dumps.

Assisting Manual Analysis While our approach is primarily designed to speed up and automate the malware family detection process, it could also be useful for manual malware analysis. Due to the small number of extracted pages, an analyst could inspect those pages manually to quickly gain additional insights about the analyzed malware. This could be useful if the analyst wants to avoid a lengthy unpacking process, since the characteristic pages should contain the unpacked code pages of the malware. One possibility would be that an analyst uses our approach and does the remaining analysis on the extracted pages using static analysis tools.

Method	TPR	FPR
Critical APIs [25]	47.56%	55.87%
peHash [26]	2.44%	0%
AVCLASS [13]	70.73%	2.36%
XOS	93%	0.15%

TABLE II
RELATED WORK COMPARISON

VI. RELATED WORK

To the best of our knowledge, we describe and release the first approach to automatically extract code signatures from packed malware by introducing the concept of cross-OS execution. While prior works already aimed to cluster/classify malware, we are not aware of a technique that can readily extract family-specific code signatures from even packed binaries. In the following, we will discuss prior classification ideas and summarize other related studies.

Malware Classification XOSSIG extracts malware fingerprints that can be used in a memory-forensic setting to classify if a malware family has infected a system. Related work addressed malware classification problems from orthogonal angles. For example, Shrestha et al. [19] and Tian et al. [20] aim to classify malware based on the strings found in the executable. While printable strings are a characteristic indicator, malware typically protects against such data signatures using on-demand data de-/encryption. Lee et al. [21] classify malware based on a sequence of API calls monitored during execution of all processes on a system, which is not possible in our case as we do post-mortem analysis. Other solutions such as by Santos et al. [22], Blichmann [23] or Bilstein et al. [24] extract byte sequences from the malware code to re-identify a family. All of these approaches either require completely unpacked samples as input or have a setup for automatically unpacking the sample that is susceptible to evasion. Apart from that, we want to emphasize again that XOSSIG targets a different use case, i.e., detecting the presence of a certain malware family in a memory dump as opposed to classifying samples. Since memory dumps are orders of magnitudes larger than samples, the chance of false positives therefore drastically increases.

More closely related to our methodology are approaches that can be used to classify the malware family based on static information and packed samples. For example, Sathynarayan et al. [25] propose classifiers based on API calls. They use a list of critical APIs and measure their call frequency to get characteristic profiles for a malware family. Wichersky [26] proposed *peHash*, which collects static information from the PE header and creates a hash of malware samples that can be used to recognize similar samples from the same family. Sebastián et al. [13] proposed AVCLASS, which labels malware as a variant of a known family by normalizing anti-virus labels. To demonstrate how their classification performance relates to ours, we tested their implementations or reimplemented the approaches in a comparative evaluation. We used our dataset (Section IV) for training and testing the related approaches.

The results of this comparison is depicted in Table II. A problem for the critical API call methodology is that the majority of the samples do not use as many critical API calls. For example, 65% of all samples in our case use only up to 5 critical APIs, which is not sufficiently characteristic to reliably identify families. The problem here is that Sathynarayanán et al. extract API call frequencies statically, whereas the low number of calls is a strong indication that the majority of the samples are packed. This is also unfavorable for peHash, which is even more sensitive to such static obfuscation techniques. The calculated hash was different for almost all malware samples, and useless for classification. This also causes peHash's FPR to drop to 0%, as we almost never saw a hash learned during the training phase again. In fact, only twice, once for Tedroo and once for Dridex, we encountered a true positive. Finally, AVclass performs the best of the three with a TPR of 70.73% and a FPR of 2.36%. Given that AVclass is based on labels of modern AV engines that can deal with standard obfuscation and packing techniques, it does not suffer to the same extent that the two other approaches do. However, AVclass's accuracy is bound by the accuracy of the AV labels, and hence, XOSSIG outperforms it.

Cross-Sandbox Execution To the best of our knowledge, we are the first to exploit OS diversity in the broader context of malware analysis. Kirat et al. [12] and Balzarotti et al. [27] proposed to execute malware in multiple sandbox environments to detect evasive malware behavior in analysis systems. Whereas we diversify the OS, these works execute malware in the *same OS* and diversify the hypervisor. They use this diversification to search for differences in the behavioral profile to detect sandbox-evasive behavior. While their methodology and goal is different from ours, their usage of cross-environment execution motivated us to explore cross-OS execution in our context.

Memory Forensics While we are not aware of works that classify malware families forensically, as described next, VMI has been used to detect suspicious malware behavior in prior work. Other approaches like *Quincy* by Barbosch et al. [28] or *ShellOS* by Snow et al. [29] try to detect malicious code in a purely forensic setting. However, they both stop at detecting malicious code of *any* malware, which is in contrast to our approach which detects a *specific* malware family and learns the corresponding code pages. Also, both Quincy and ShellOS detect very specific strains of malicious code, i.e., code injections and shellcode, which in turn requires strong domain knowledge which our approach does not.

Malware Unpacking and Code Extraction An interesting aspect of our work is that by extracting the characteristic pages we implicitly unpack the malware. While the unpacking is not complete (lack of coherent buffer of bytes, loss of PE headers, etc.), our approach may assist analysts in unpacking. Compared to other solutions [30], [31], [32], our approach benefits from the fact that it does not require any domain-specific or OS modifications for code extraction.

Code Identification For identifying the code of the malware, we use a custom dissassembly-based heuristic. This is different from what other research on this topic has used so far. Irfan et al. [33] propose to use the relocation information of executable files to solve this problem, which however we cannot rely upon for stripped and packed malware executable. *ByteWeight* by Bao et al. [34] or *Nucleus* by Andriesse et al. [35] aim to solve the problem of finding functions in code. Whereas ByteWeight uses machine learning to learn function prologues and epilogues, Nucleus is based on CFG analysis. However, since the objective of those approaches is not to detect *code*, but rather to find the starting and ending offsets of functions *inside* already-identified code, these approaches do not solve our core problem of identifying code.

VII. Conclusion

We introduced *Cross OS Execution* to automatically learn the forensic fingerprint of malware by running a malicious sample on different OSes. Compared to other solutions, our method benefits from complete automation, is agnostic of OS-specific idiosyncrasies, and raises the bar for evasion given the generic nature of our approach. Our research stresses the importance of certain aspects in the area of malware classification that have not been targeted by most academic work so far. We experimentally verified that we can (i) *automatically learn* the characteristics of a malware family, (ii) use those characteristics to *automatically detect* them in infected memory dumps, and (iii) that all of this can be done in a *purely memory-forensic* setting. Our results show that it is feasible to focus on *pure code* to create characteristic malware fingerprints. We believe that defenders can significantly extend their knowledge of malware by adopting ideas of our approach and hope to foster further research in this area with this work.

References

[1] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust Signatures for Kernel Data Structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[2] B. Stock, B. Livshits, and B. Zorn, "Kizzle: A Signature Compiler for Detecting Exploit Kits," in *The 46th Annual IEEE/IFIP Conference on Dependable Systems and Networks*. IEEE, 2016.

[3] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking Rootkit Footprints with a Practical Memory Analysis System," in *Proceedings of the 21st USENIX Security Symposium (USENIX SEC)*, 2012.

[4] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers." in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[5] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using System-centric Models for Malware Protection," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[6] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov, "Botzilla: detecting the "phoning home" of malicious software," in *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, 2010.

[7] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic Analysis of Malware Behavior Using Machine Learning," *Journal of Computer Security*, vol. 19, no. 4, Dec. 2011.

[8] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proceedings of the 18th USENIX Security Symposium (USENIX SEC)*, 2009.

[9] GlobalStats, "Desktop Windows Version Market Share Worldwide." [Online]. Available: http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide

[10] Capstone, "The Ultimate Disassembly Framework - Capstone - The Ultimate Disassembler." [Online]. Available: https://www.capstone-engine.org

[11] 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)*, 2008.

[12] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal Analysis-based Evasive Malware Detection," in *Proceedings of the 23rd USENIX Security Symposium (USENIX SEC)*, 2014.

[13] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A Tool for Massive Malware Labeling," in *Proceedings of the 19th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2016.

[14] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2007.

[15] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM Monitoring Using Hardware Virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[16] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[17] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.

[18] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating Code from Data in x86 Binaries," in *Proceedings of Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 2011.

[19] P. Shrestha, S. Maharjan, G. R. de la Rosa, A. Sprague, T. Solorio, G. Warner, and K. Pichara, "Using String Information for Malware Family Identification," in *Advances in Artificial Intelligence*, 2014.

[20] R. Tian, L. Batten, R. Islam, and S. Versteeg, "An Automated Classification System Based on the Strings of Trojan and Virus Families," in *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[21] J. Lee, K. Jeong, and H. Lee, "Detecting Metamorphic Malwares Using Code Graphs," in *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, 2010.

[22] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-Sequence-Based Malware Detection," in *Engineering Secure Software and Systems: Second International Symposium, ESSoS*, 2010.

[23] C. Blichmann, "Automatisierte signaturgenerierung f"ur malware-sta"mme," 2008.

[24] F. Bilstein, "YARA-Signator: Automated Generation of Code-based YARA Rules," 2019. [Online]. Available: https://www.botconf.eu/wp-content/uploads/2019/12/B2019-Bilstein-Plohmann-YaraSignator.pdf

[25] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature Generation and Detection of Malware Families," in *Proceedings of Information Security and Privacy, 13th Australasian Conference (ACISP)*, 2008.

[26] G. Wicherski, "peHash: A Novel Approach to Fast Malware Clustering," in *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET)*, 2009.

[27] D. Balzarotti, M. Cova, C. Karlberger, K. Christopher, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[28] T. Barabosch, N. Bergmann, A. Dombeck, and E. Padilla, "Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[29] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks," in *Proceedings of the 20th USENIX Security Symposium (USENIX SEC)*, 2011.

[30] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "RAMBO: Run-Time Packer Analysis with Multiple Branch Observation," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[31] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[32] W. T. Strayer, D. Lapsely, R. Walsh, and C. Livadas, "Botnet Detection Based on Network Behavior," in *Botnet Detection: Countering the Largest Security Threat*. Springer, 2008.

[33] A. Irfan, V. Roussev, and A. A. Gombe, "Memory Forensics: Reliable In-Memory Code Identification Using Relocatable Pointers," in *Proceedings of the 67th Annual Meeting of the American Academy of Forensic Sciences (AAFS)*, 2015.

[34] R. Bao, Tiffany Wang, Y. Shoshitaishvili, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium (USENIX SEC)*, 2014.

[35] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-Agnostic Function Detection in Binaries," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.