MutantX-S: Scalable Malware Clustering Based on Static Features

Xin Hu, Kang G. Shin
University of Michigan, Ann Arbor

Sandeep Bhatkar, Kent Griffin
Symantec Research Labs

ABSTRACT

The current lack of automatic and speedy labeling of a large number (thousands) of malware samples seen everyday delays the distribution of malware signatures, leading to a low detection rate of new malware samples in the wild. In this paper, we design, implement and evaluate a novel, scalable framework, called MutantX-S, that can efficiently cluster a large number of samples into families based on programs' static features, i.e., code instruction sequences. MutantX-S is a unique combination of several novel techniques to address the practical challenges of malware clustering. Specifically, it exploits the instruction format of x86 architecture and represents a binary program as a sequence of opcodes, facilitating the extraction of N-gram features. It also exploits the hashing trick recently developed in the machine learning community to reduce the dimensionality of the extracted feature vectors, thus significantly lowering the memory and computation costs of clustering. Our comprehensive evaluation on a MutantX-S prototype using a database of more than 100,000 malware samples has shown its ability to correctly cluster over 80% of input samples within 2 hours, achieving a good balance between accuracy and scalability. Applying MutantX-S on malware samples created at different times, we also demonstrate that MutantX-S achieves high accuracy (around 0.75–0.8) in predicting family labels for unknown malware.

1. INTRODUCTION

According to the Symantec's latest Internet Threat Report, 403 million new variants of malware were created in 2011, a 41% increase from 2010. This exponential growth of malware samples has created a major challenge for anti-virus (AV) companies: how to efficiently process this huge influx of new samples and accurately labels them? Typically, AV companies receive several thousands of suspicious samples everyday. It is practically impossible to manually analyze such a huge number of samples, leaving a large fraction of samples unlabeled and hence delaying the signature distribution. One possible solution to this is to *automatically* cluster malware samples and assign them labels according to their similarities. The intuition behind this is that malware programs bearing significant similarities are likely to have been derived from the same code base, and hence from the same malware family. One can thus group similar samples into a cluster and label them with high accuracy by analyzing only a few representative samples. Moreover, the label of a new sample can be automatically derived if it is determined to belong to an existing cluster. In this paper, we design, implement and evaluate MutantX-S, a novel and scalable system, that can efficiently cluster a large number of malware samples into families based on their static features, i.e., code instruction sequences.

Most existing malware clustering/classification systems are based on dynamic behavioral features. These dynamic-analysis systems operate by running malware samples in virtual or sandboxed environments, monitoring their execution and extracting their run-time behavior in terms of API or system call traces [5, 6, 22]. The major benefit of using dynamic behavioral features is that they are less susceptible to mutation schemes frequently employed by malware writers to avoid static analysis, e.g., such as run-time packers or binary obfuscation. Albeit very useful in practice, approaches based on dynamic behavioral features also suffer from several limitations. First, they may have only limited coverage of an application's behavior, failing to reveal the entire capabilities of a given malware program. This is because for a particular execution run, a dynamic analysis can only capture API or system call traces corresponding to the code path that was taken during that particular execution. Different code paths may be taken in different runs, depending on the program's internal logics and/or external environments. Also, malware often include triggers in their programs and exhibit an interesting behavior only when certain conditions are met. Typical examples include bot programs that wait for commands from their botmasters, and malware programs designed to launch attacks on a certain date. These trigger-based malware generate very few run-time traces and thus cannot be captured by dynamic analysis. Second, dynamic analysis is inherently resource-intensive and hence doesn't scale well. To process the sheer number of malware samples collected everyday with limited resource, a dynamic-analysis system can execute and monitor each sample only for a short period of time, e.g., a couple of minutes. Unfortunately, this time is often too short for typical malware programs to reveal all their true behavior.

In this paper, we present MutantX-S, a new and practical system that exploits static features of code instruction sequences for efficient and automatic malware clustering and labeling. MutantX-S is motivated by the common observation that a large portion of today's malicious programs are file-level variations of a small number of malware families and tend to share similar instruction sequences in their binaries. Analysis of static features of malware offers several unique benefits. First, it has the potential to cover all possible code paths of a (malicious) program, yielding more accurate representations of the entire functionalities of the programs. Moreover, approaches based on static features are much more scalable than their dynamic counterparts, as they do not require resource-intensive and time-consuming monitoring of malicious programs. This is particularly important for AV companies to process a rapidly-increasing number of new malware samples. Unfortunately, the static-feature-based approaches are not without limitations of their own. It is well-known that they suffer from run-time packing and obfuscation. Therefore, the goal of `MutantX-S` is not to replace existing dynamic-behavior-based systems, but to complement them to achieve higher clustering accuracy and better coverage of malware programs.

`MutantX-S` features a unique combination of techniques to address the deficiencies of static malware analysis. First, it employs an efficient encoding mechanism that exploits the IA32 instruction format to encode malware binaries into a opcode sequence, facilitating the extraction of N-gram features. Second, it applies a hashing-trick on the extracted N-gram features that help the clustering algorithm handle very high dimensional features. Finally, it tailors a generic unpacking technique to handle commonly-seen run-time packers so that the clustering algorithms may be applied to a larger set of malware samples. We have successfully implemented a fully-automated prototype of `MutantX-S` and evaluated its performance using a database of more than 100,000 distinct malicious programs. Our evaluation results demonstrate that `MutantX-S` can effectively create clusters corresponding to malware families which allows accurate prediction of new malware labels and reduce/remove the manual analysis effort, thereby enabling faster response to new malware threats.

The rest of the paper is organized as follows. Section 2 surveys previous work related to malware analysis. Section 3 briefly describes the architecture of `MutantX-S` followed by elaboration of each component of `MutantX-S` including the unpacking (Section 4), instruction feature extraction (Section 5) and clustering (Section 6). The comprehensive performance evaluation of `MutantX-S` is presented in Section 7. Section 8 discusses the limitation and potential improvement of `MutantX-S`, and Section 9 concludes the paper.

2. RELATED WORK

Malware pose one of the severest threats to computer systems and the Internet. As a result, automatic malware clustering and classification have recently attracted considerable attention. Various schemes have been proposed to tackle this problem based on the dynamic behavior and static features of malware.

Dynamic-analysis approaches have the major benefit of handling packed and obfuscated malware samples by executing malware programs in a virtual or sandboxed environment and collecting their behavior in terms of system or API calls. Lee and Mody [16] proposed use of a sequence of events (e.g., registry and file system modifications) to cluster similar malware programs and assigned the same class label to a new malware sample as that of its nearest neighbor in a set of known samples. Rieck *et al.* [21] converted the frequency of run-time behavior, such as copy file and create processes, into feature vectors and applied SVM (Support Vector Machine) to learn and classify unknown samples to their closest kin. One limitation of these approaches, as noted in [6], is that it uses supervised learning techniques and thus requires labeled training sets. Later, Bailey *et al.* [5] applied a hierarchical clustering algorithm to group similarly-behaving malware samples. Unfortunately, the complexity of this clustering algorithm is $O(n^2)$, limiting its applicability only to a small number of samples. To address this problem, Bayer *et al.* [6] and Rieck *et al.* [22] developed different methods to scale the clustering algorithms. Bayer *et al.* [6] applies locality-sensitive hashing (LSH) to efficiently compute an approximate hierarchical clustering with a significantly smaller number of distance computations. By contrast, Rieck *et al.* [22] developed a prototype-based clustering algorithm that reduces the runtime complexity by performing clustering only on representative samples (i.e., prototypes). Comparing to LSH clustering, a prototype-based algorithm facilitates the analysis of behavior groups because each prototype represents a particular malware group [22]. In `MutantX-S`, we adopt the same prototype-based algorithm as in [22] because of its efficiency and explicit expression of malware features.

Static analysis, on the other hand, uses features extracted directly from malware binaries as the basis for analysis. Christodorescu *et al.* [7] extracted unique malicious patterns from disassembled malware that are resilient to obfuscation. Wicherski [28] utilizes static features from PE headers, e.g., raw size, entry point, import table, etc., to group similar malware. Karim *et al.* [12] took a different approach and studied the malware evolution by creating phylogeny models of malware families based on N-gram and N-perm on assembly instructions. Similar features have also been used in [14] to validate various supervised learning methods, such as naive Bayes, decision trees, SVM, etc. `MutantX-S` falls into the static-analysis category since it relies on (malware) features extracted from the malware code instructions. The main difference of `MutantX-S` from previous approaches is its unique combination of techniques that ensures the scalability to large malware datasets. Although malware sets of a similar size have been studied with dynamic-behavior-based clustering [6], static analysis is still necessary and sometimes more advantageous since it does not suffer from the limited coverage of dynamic analysis. Another system similar to `MutantX-S` is BitShred [11] which also focuses on the scalability of malware comparison and triage on a large scale. BitShred developed a fast code-comparison algorithm based on hashes of byte sequences in code section and made use of distributed computing resources, i.e., Hadoop and MapReduce to achieve a high throughput in binary comparison and good scalability.

3. ARCHITECTURE

Figure 1 shows an overview of `MutantX-S`. At a high level, `MutantX-S` takes a set of malicious or suspicious program samples as input and extracts their features using static analysis to avoid the computational overhead and maximize code coverage. Specifically, `MutantX-S` first uses existing tools (e.g., PeID [1], TrID [19], SymPack[1]) to identify malware files that have been processed by the binary packing/encryption tools such as UPX [26], ASPack[4], and other customized packers. These files will be unpacked with a generic unpacking technique tailored for `MutantX-S`. Together with samples that are in their original binary (not packed), they are disassembled to extract their code instructions. These pre-processing steps ensure that `MutantX-S` can successfully extract the features inherent to malware families without influence of encryption or compression. After their pre-processing, all malware samples are passed to the second component and processed with three steps to extract their representative features: (1) *Instruction Encoding* for converting each instruction to a sequence of encoded operation codes that capture the underlying semantics of the programs, (2) *N-gram analysis* for constructing feature vectors that allow computation of program similarities, and (3) *Hashing Trick* for compressing the feature vectors, significantly improving the speed of similarity computation while incurring only a small penalty in clustering accuracy. Finally, a prototype-based clustering algorithm is applied on the set of compressed feature vectors and partitions samples into different clusters, each representing a group of similar malicious programs.

4. GENERIC UNPACKING ALGORITHM

Due to its simplicity and effectiveness, run-time packing is one of the most commonly used techniques by malware writers to circumvent anti-virus detection. More than 80% of malware programs are estimated to be packed by some type of packers [9]. A typical packer like UPX works as follows. UPX first compresses all the code and data sections of a portable executable (PE) binary[2] into a single section. Then, it creates a new PE binary containing the compressed data followed by the unpack code. The entry point in the new PE header is altered to point to the unpacker code such that the unpack code will first be executed when the packed program runs. The unpack code decompresses the original program codes into memory and then jump to the first instruction of the restored codes (i.e., the original entry point) to resume execution. This packing process enables malware programs to disguise their malicious instructions as random-looking data while keeping the original functionality intact. Since all static analysis tools including `MutantX-S` rely on features extracted from original instructions, it is imperative for `MutantX-S` to handle packing correctly and efficiently.

While there exist unpacking tools such as UPX itself, ArmaGedoon, etc., they are often targeted specifically at one or a few packers. As more packers appear in the wild, the cost of manually reverse-engineering packers and continually updating unpacking tools is expected to grow over time. In addition, unpacking tools often have to perform expensive processing to ensure that the unpacked program can be successfully executed (e.g., the PE headers must be correctly specified and import table has to be reconstructed), making them too slow to be used for processing large number of malware samples. `MutantX-S` , on the other hand, has no need to guarantee the executabilty of the unpacked programs as long as the original instructions can be inspected and features extracted. `MutantX-S` thus tailors a generic unpacking mechanism to meet the particular need for malware clustering and improves its efficiency. The basic idea is to exploit the inherent property of the unpacking procedure, i.e., a packed binary has to write the unpacked code into some memory space and transfer control to the modified memory locations to continue execution. By continual monitoring of memory access, we can learn that some form of unpacking, self-modification or on-the-fly code generation instructions occurs if the program execute a memory address after writing something into the address. These written-then-executed memory locations are likely to contain the original program codes and thus are the targets to be analyzed by `MutantX-S`.

The unpacking component of `MutantX-S` exploits the physical non-execution (NX) support in modern x86 CPUs and Windows OS to track memory page status. It consists of a kernel driver responsible for tracking system calls and a user-level component that is injected as a remote thread into a program's address space. The unpacking component of `MutantX-S` does two things: (1) runs the packed binary and dumps the

[1]An in-house tool developed inside Symantec

[2]PE is the executable file format used by the Windows Operating Systems

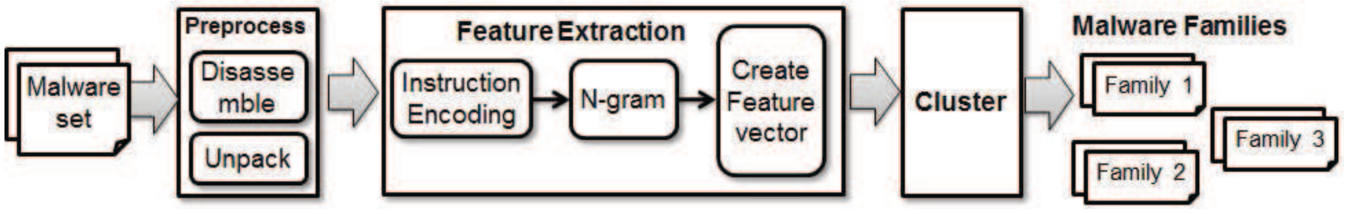


Figure 1: A system overview of `MutantX-S`

memory image of the running process at an appropriate time when the binary is likely to finish unpacking (hence the dumped memory contains the original instructions), and (2) determine the correct original entry point (OEP) for the dumped image. Finding the correct OEP is critical for program disassembly and feature extraction, because a wrong entry point may cause a disassembler to miss all the instructions between the original and the misidentified entry points (if there is no other reference to this portion of code). In addition, if the entry point is incorrectly set in the middle of an instruction, the disassembler will fail or generate completely wrong assembly codes.

The details of the unpacking process are summarized here:

1. `MutantX-S` loads the packed program into memory, suspends its execution and inject the user-level hooking DLL into the process' memory space. It marks all the memory pages of the loaded program as *executable* but *non-writable*, and resumes its execution.

2. During the execution, when the unpacker attempts to write unpacked codes into memory, a write exception will occur on the *non-writable* page. `MutantX-S` marks the page as dirty and changes its permission to *writable* but *non-executable*.

3. When the unpacker jumps to the the newly-generated code for execution (e.g. after finishing unpacking), the *non-executable* permission on these pages causes an execution exception. `MutantX-S` intercepts such exceptions and records memory addresses where the exceptions had occurred. For simple unpackers such as UPX that first unpack the entire program and then jump to the restored program for execution, the memory address where the exception occurs is the OEP (original entry point) that we want to identify. However, this is not necessarily true for more sophisticated packers (e.g., self-modifying code that may rewrite to the same memory location). Therefore, `MutantX-S` removes the write permission from these memory pages again, grants execution privilege and continues execution. `MutantX-S` also monitors dynamic allocation of memory pages and removes their write permission to track unpacking on these pages.

4. `MutantX-S` dumps the process memory image either at the end of program execution after a certain period of time. The basic intuition behind this is that after the program has been running for a sufficient amount of time (e.g., 30 seconds to 1 minute), it is fairly safe to assume that the program has finished unpacking and the original codes are contained in the memory.

Attaching a new PE header to the dumped memory image, `MutantX-S` creates a valid PE file from which a standard disassembler can disassemble instructions. However, as mentioned earlier, the major challenge for creating a valid PE file is to identify the correct value of the entry point in the PE hader. Previous unpacking tool often use the same $W \oplus X$ policy to address this problem by assuming that the packer will unpack the entire program in the memory and then execute the newly generated instructions. As a result, the entry point is simply the start address of the dirty memory page where the first execution exception occurs. Unfortunately, this is only true for simple packers like UPX. As adversaries become increasingly sophisticated, various evasion schemes have been developed to complicate the detection of OEP. A typical method is to fake end-of-unpacking by writing rouge instructions into a reserved memory page, transfer control to it, and jump back to the unpacker code. This creates an illusion to unpacking tools that unpacking has ended, and concludes with a wrong address for the entry point. More advanced unpackers uses incremental unpacking that decrypts only part of the payload and executes them before decrypting more instructions. In such cases, detecting the first execution exception is not enough because only part of the original program is visible. To address these problem, `MutantX-S` develops a new heuristic called LMFE (*Last Modification First Execution*).

The idea is to keep track of time when the last write exception and a subsequent execution exception occur on each memory page, so `MutantX-S` can identify the unpacker's attempts to write to the same memory page multiple times, in which case, the previous modification and execution on the page are likely to be spurious. More specifically, for each memory page, Mu-

Algorithm 1 `MutantX-S` unpacking algorithm

```
 1: Input:     A packed binary program $B$
 2: Output:    a reconstructed PE file containing unpacked program codes
 3: STEP 1:
 4: Load the packed program into memory
 5: for all $p$ in the program's memory pages do
 6:     $Permission(p)| = \tilde{W} // removewritepermission$
 7: end for
 8:
 9: STEP 2:
10: while $B$ is running and $T_{runtime} < T_{thresh}$ do
11:     $a$: The address of the page fault
12:     $t$: The page fault type $t \in$ {WRITE, EXECUTE}
13:     $p \leftarrow Page(a)$
14:     if t = WRITE then
15:         $Permission(p)| = (W|\tilde{X})$ // Writable but non-executable
16:         $last\_written(p) \leftarrow$ current time
17:     end if
18:     if t = EXECUTE then
19:         $Permission(p)| = (\tilde{W}|X)$ //non-writable but executable
20:         $last\_exec(p) \leftarrow$ current time
21:         $addr\_exec(k) \leftarrow a$
22:     end if
23: end while
24:
25: STEP 3:
26: Dump process memory
27: reconstruct $B'$ by setting OEP to be $addr\_exec(k)$ where:
28: $k = \arg\min_k(last\_exec(k) > \max(last\_written(i))$
29: return $B'$
```

tantX keeps a record of it 1) last modification time (i.e., a write exception occurred on the page), denoted as *last_written*; 2) last execution exception time, denoted as *last_exec*; and 3) the address *addr_exec* where the exception had occurred. At any point of execution, there are three types of memory pages.

Type I: memory pages that have valid *last_written* and *last_exec*, i.e., pages that are both modified and executed.

Type II: memory pages that have valid *last_written* but not *last_exec*, i.e., pages that are modified but not executed. They could either be page containing pure or code pages that have not yet been executed.

Type III: memory pages that have neither valid *last_written* nor valid *last_exec*. These could be initialized data-section pages or unpacker-code pages.

Essentially, type-I memory pages are those that hold the unpacked instructions and thus contain the OEP. When dumping the process memory, `MutantX-S` uses the following algorithm to pinpoint the correct OEP. Let $P(i),\ i = 1..n$ represent all type-I memory pages of the packed program and $last\_written(i)$, $last\_exec(i)$ and $addr\_exec(i)$ represent the time of the last write exception, last execution exception and address where the exception occurs for page $P(i)$, respectively. Then, the OEP is $addr\_exec(k)$ in the memory page $P(k)$ where

$$k = \arg\min_k(last\_exec(k) > \max(last\_written(i)) \quad (1)$$

where $i = 1..n$. In other words, $P(k)$ is the first memory page that is executed after all type-I memory pages have been written. Below we show that `MutantX-S` is able to find the correct OEP (i.e. $addr\_exec(k)$) even when the packers try to fool the `MutantX-S` using spurious write-and-execute sequences or multi-layer packing.

Proposition. *For k satisfying Eq. (1), $addr\_exec(k)$ is the correct OEP of the original program no matter whether the program is packed with simple packers or more advanced packers that fake end of unpacking.*

Proof. Let us first look at a simple packer like UPX that restores the entire program into memory before transferring control to it. Let $P(j),\ j = 1..m$ denote memory pages where the unpacker writes the original program after decryption. Without loss of generality, we can assume that the contents are written sequentially from $P(1)$ to $P(k)$, meaning that $last\_written(1) < last\_written(2) < \ldots < last\_written(m)$. When the packer finishes unpacking and starts executing the restored program by jumping to the OEP, an execution exception will first occur on the memory page $P(k)$ that contains the OEP, i.e., $last\_exec(k) > last\_written(m)$ and $last\_exec(k) < last\_exec(j) \forall j \neq k$. As a result, $addr\_exec(k)$ is the correct OEP.

Second, assume a more advanced packer with the following spurious unpacking sequence: it writes arbitrary instructions into some memory page, executes them and, at the end of execution, returns to the unpacker code. Such a routine may be called multiple times during the whole unpacking process. As a result, an unpacking tool will fail if it assumes the end-of-unpacking at the first (or first few) execution exception. `MutantX-S` is resilient to this type of evasion by enforcing the invariant that the execution exception on the OEP must succeed all the write exceptions. For example, when the spurious unpacking routing touches memory page $P(s)$, `MutantX-S` records $last\_exec(s)$ and marks $P(s)$ as executable but non-writable. Then, the unpacker resumes the normal unpacking and writes more decrypted instructions to memory page $P(t)$ (t could be any memory page including s). This creates a new write exception on $P(t)$ at timestamp $last\_written(t)$. Note that because $last\_exec(s) < last\_written(t)$, `MutantX-S` determines s to not contain the OEP. In contrast, after the packer finishes unpacking and transfers control to the real OEP, the execution exception satisfies Eq. (1). By keeping *addr_exec* up-to-date and pointing to a valid instruction, `MutantX-S` is able to recognize the real OEP accurately. Same arguments hold for multi-layer packing because the write exceptions of code pages will always precede the executable exceptions caused by jumping to the OEP. □

With the correct OEP identified, MutantX-S attaches a new PE header to the dumped the memory images. The correct OEP ensuring a proper starting point for the disassembler and the dumped memory image contains the original unpacked programs. The whole process is summarized in Algorithm 1.

5. FEATURE EXTRACTION

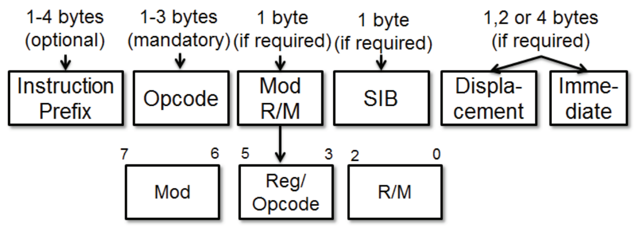


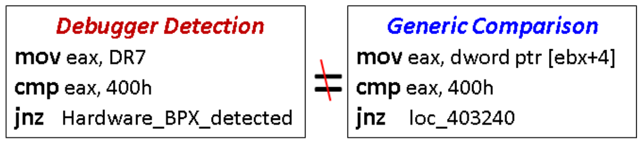
Figure 2: x86 instruction format

Given the reconstructed (unpacked) malware programs, MutantX-S uses the IDA Pro disassembler [3] to disassemble them down into a sequence of machine instructions that are then used for feature extraction. The key step in MutantX-S is the comparison of similarities between malware samples based on the disassembled instruction sequences (e.g., Move EAX, EBX, cmp eax, 1, ..). The main challenge in similarity comparison lies in handling the variations of machine instructions. Malware often undergo changes for many reasons, such as mutation, polymorphism, and obfuscation where instruction sequences that are semantically equivalent are used to replace each other. Hence, ensuring exactness in comparing instructions will not tolerate any variation in the syntax. At the other extreme, correctness is compromised if all forms of variation are tolerated. MutantX-S strikes a balance between these two extremes by exploiting the x86 instruction format (Fig. 2) and uses the opcode as a succinct representation of the instruction semantics.

Using opcodes—instead of other features used in previous work such as control flow graphs, binary sequences or mnemonic sequences—offers several benefits. First, opcodes generalize well to represent variants of a malware family. Malware samples in the same family are often derived from the same code base and thus share similarities in their instructions. However, due to relinking, rebinding and rebasing, the operands (e.g., registers, memory addresses) of instructions tend to change across the variants. Using opcodes and ignoring the operands (i) make MutantX-S more resilient to low-level mutations while providing a meaningful characterization of semantics and (ii) reflect the functionality of the malware programs. Second, observing the variability of operands, previous approaches instead use mnemonic sequences (e.g., mov, push) to represent the instruction functionalities. When testing MutantX-S, we discover that the opcode sequence offers better representation of instruction semantics. Mnemonics sometimes *overly* generalize the underlying CPU operations, causing many different instructions (or instructions with distinct semantics) to appear similar. To illustrate this, consider all the instructions in Table 1. Although all of them have the same mnemonic (i.e., mov), the underlying functionalities are drastically different. For instance, moving a value to a control register or debug often indicates a critical OS operation, such as interrupt control, switching addressing mode or enable/disable debuggin, etc., which should not be treated same as moving a value between one register and another. Ideally, moving data from memory to a register (memory load operation) should be considered as a distinct operation from that of moving from a register to memory (memory store operation) too. Unfortunately, using mnemonics would cause all these distinct instructions to be represented with a single feature (i.e., mov), which may lead to an accidental similarity between code sequences. As illustrated in Figure 3, On the other hand, features based on opcode provide higher distinguishability between semantically different instructions and thus better accuracy in clustering.

Opcode	Instruction	Description
89	MOV r/m32, r32	Move from reg to mem/reg
8B	MOV r32, r/m32	Move from mem/reg to reg
B8	MOV r32, imm32	Move immediate val to reg
0F 20	MOV r32, CR0-CR4	Move from control reg to reg
0F 22	MOV CR0-CR4,r32	Move from reg to control reg
0F 21	MOV r32, DR0-DR7	Move from debug reg to reg
0F 23	MOV DR0-DR7,r32	Move from reg to debug reg

Table 1: Opcodes provide fine-grained representations of instruction semantics (reg: register, mem: memory)


Debugger Detection
≠
Generic Comparison

```
mov eax, DR7
cmp eax, 400h
jnz   Hardware_BPX_detected
```

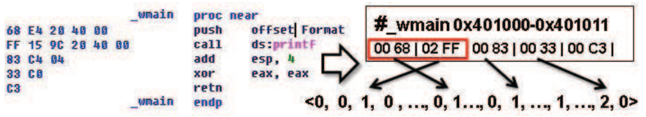
```
mov eax, dword ptr [ebx+4]
cmp eax, 400h
jnz   loc_403240
```

Figure 3: Two code pieces with completely different semantics share same mnemonic representation (i.e., move, cmp, jnz). However, they can be differentiated by their opcode representation: "0F 21 3D 75" vs "8B 3D 75"

With this encoding scheme, a program can be represented as a sequence of encoded opcodes (Fig. 4). MutantX-S then uses the standard N-gram analysis to characterize the content of a malware program, i.e., moving a fixed-length window over the sequence and

[3]the de-facto disassembler for the analysis of hostile code

consider a subsequence of length N at each position. The resulting N-gram of opcodes reflects short instruction patterns and implicitly captures the underlying program semantics. Then MutantX-S constructs a feature vector vector V in an $|S|$-dimensional vector space ($|S| = |\mathcal{O}|^N$ where $\mathcal{O}$ is the set of all possible opcodes) . Each dimension of V is the number of *occurrence* of one particular opcode N-gram. Then MutantX-S can geometrically caculate the similarity between two malware programs (m, v) as the Euclidean distance between their feature vectors in the vector space: $d(m,n) = \|V_m - V_n\| = \sqrt{\sum_{i=1} |S|(V_m(i) - V_n(i))^2}$. Compared to the other similarity metrics (e.g., locality-based hashing), geometric assessment of similarity in the vector space provides the benefit of *explicit feature representation* [22] where the importance or contribution of each N-gram in clustering similar malware can be traced back to its original code patterns. For N-grams that potentially correspond to inherent characteristics of a malware family, e.g., those that frequently appear within a family but rarely occur in others, their original code segments can be traced back and used signatures to detect malware variants.



```
                 _wmain   proc near
68 E4 20 40 00            push    offset Format
FF 15 9C 20 40 00         call    ds:printf
83 C4 04                  add     esp, 4
33 C0                     xor     eax, eax
C3                        retn
                 _wmain   endp
```

#_wmain 0x401000-0x401011
00 68 | 02 FF 00 83 | 00 33 | 00 C3 |
<0, 0, 1, 0 , ..., 0, 1..., 0, 1, ..., 1, ..., 2, 0>

Figure 4: Encoding a function into a feature vector

6. CLUSTERING ALGORITHM

Next step in MutantX-S is clustering malware samples into groups that share common traits. Considering the enormous number of malware in the wild, the goal of MutantX-S is to process hundreds of thousands malware files sufficiently fast. Unfortunately, classic clustering algorithms such as hierarchical and partitioning-based clustering, e.g., K-Means or K-Medoids—although they have been successfully applied to cluster malware behavior [5] and create phylogeny trees [12]—incur a time complexity at least quadratic in the number of samples, which in practice, does not scale to the MutantX-S' target. MutantX-S exploits two approaches to address the scalability issue: (1) a hash kernel that compresses the high dimensional feature vector into a low dimensional space, and (2) a prototype-based clustering algorithm that has close-to-linear runtime complexity.

6.1 Hashing Kernel

Kernel methods [23] are powerful tools used in machine learning to allow operation in the high-dimensional feature space without ever having to compute the coordinates of the data in that space. This is particularly useful when the input data has a non-linear decision boundary but can be linearly separated in a high dimensional feature space. In MutantX-S, however, we have encountered the opposite problem: the original space is very high-dimensional[4]. The number of dimensions D determines the complexity when computing the vector distance and D increases exponentially with N in N-gram (i.e. $D = |\mathcal{O}|^N$ where $|\mathcal{O}|$ is the number of different opcodes and in practice $|\mathcal{O}| > 200$). Therefore even a small N like 3 will result in a (very sparse) feature vector with more than 8 million dimensions, which is computationally prohibitive when calculating similarities for large number of malware samples. Unfortunately, N has to be at least 3 or 4 to be descriptive enough for capturing the program semantics.

To address the problem, MutantX-S exploits the *hashing-trick* recently developed in the machine learning community[24], which *hashes* the high dimensional input vector $x \in \mathbb{R}^n$ into a lower dimensional feature space $\mathbb{R}^m$ with the mapping function $\phi : \mathcal{X} \to \mathbb{R}^m$. Since $m \ll n$, the hashing trick reduces a feature vector to a more compact representation, allowing the clustering algorithm to handle a large volume of data and save both computation and memory requirements. Previous research has shown that the hash kernel approximately preserves thevector distance, because the penalty incurred from using a hash for reducing dimensionality only grows *logarithmically* with the number of samples and groups [24, 13].

Specifically, MutantX-S applies a *uniform* hash function H : {N-gram} $\to [1..m]$ that hashes N-gram directly into a position in the feature vector of length m. In case of a collision where two or more N-grams map to the same position, the sum of their counts is used as the value in the new vector. More formally, for malware M and M', let v and v' represent their original feature vector extracted from the encoded opcode sequences and ξ denote the mapping from the N-gram $(o_1, o2, \ldots, o_N) \in S$ to the index in v. We define the hash feature map ϕ as

$$\phi_i(v) = \sum_{l:H(o)=i, l \in S} v(\xi(o))$$

and the distance between M and M' as

$$d_\phi(M, M') = \|v - v'\|_\phi = \|\phi(v), \phi(v')\|.$$

The choice of m, the length of the low dimensional vector, is a tradeoff between clustering accuracy and storage overhead as well as computation complexity. Choosing a smaller m means shorter vector length, thus, fast distance computation and smaller memory footprint to malware features. However, decreasing m reduces the number of bins in which the hash function can place the different N-gram and consequently increases

[4]thus, the input data are likely already linearly separable

the collision possibility, leading to over-compression of features and negative impact of the clustering accuracy.

6.2 Prototype-Based Clustering

The majority of computation in clustering goes to the comparison of malware samples. Classic clustering algorithms typically incur a complexity that is super-linear in the size of the input data. For example, the running time for two most widely-used clustering algorithms k-means and hierarchical clustering are $O(n^{kd})$ [3] and $O(n^2 \log n)$ [17], resulting in the computation time that is prohibitively large for the number of malware samples we have to deal with. To address this scalability problem, `MutantX-S` adopts the prototype-based linear-time clustering algorithm designed in [22].

Prototype-based algorithms belong to the type of unstructured and model-free methods for clustering and pattern matching. Despite their simplicity, they have been empirically shown to be very effective and often one of the best performers in real data [10]. Prototype-based clustering first extract a set of prototypes each of which serves as the representative for a small group. Then the remaining data points are associated with their closest prototype in the feature space. The key idea of Prototype-based algorithm is to perform computation (e.g. clustering) only on the prototypes which are a small subset of original data points, thus reducing the computation time significantly. The algorithm consists of following two steps.

Prototype extraction: The quality of final clusters depends on the choice of the prototypes. Well-positioned prototypes can accurately capture the distribution of input data and allows creating accurate class boundaries in the feature space. Unfortunately, determining the optimal number and positions of prototypes has been shown to be NP-hard [22]. For scalability consideration, an approximate algorithm by Gonzàlez [8] was commonly used to iteratively select prototypes. During each iteration, the data point with the largest distance to existent prototypes is selected as the next prototype[5]. The process is is repeated until the distance from all the data points to their nearest prototype is smaller than a predefined threshold P_{max}. In other words, all the data points are located within a certain radius from their closest prototypes. The run-time complexity of this algorithm is $O(kn)$ where k is the number of prototypes selected. Since k only depends on the distribution of the data (in this case, k is proportional to the number of similar malware groups or families), with a reasonable choice of P_{max} the algorithm is linear in the number of input data n.

Clustering with prototypes. Instead of working on the huge number of original data points, the algorithm performs agglomerative hierarchical clustering only on the prototypes. Specifically, the algorithm starts with individual prototypes as singleton clusters, successively merges two closest clusters, and terminates when the distance between the closest clusters is larger than a predefined distance threshold Min_d. Then, prototypes within the same cluster are assigned the same cluster label and subsequently propagate the label to their associated data points. Because each prototype is a good representation of its associated data points (all within a radius of P_{max}), the algorithm avoids expensive distance computation between the original data points without too much loss in the overall accuracy. The respective run-time complexities of clustering and propagation steps are $O(k^2 log\ k)$ and $O(n)$. Compared to the $O(n^2 log\ n)$ complexity of applying an hierarchical clustering algorithm on the original data points, this algorithm achieves a significant speed-up, with a factor of at least $(n/k)^2$.

[5]the first prototype is selected randomly

7. EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency and accuracy of `MutantX-S` based on two data sets: (1) a reference data set containing 4821 malware files whose labels are generated by security experts from a large anti-virus company and thus more reliable; and (2) a large malware data set collected from an online malware archive [27] which comprises 132,234 malware samples with unreliable labels derived from AV scanners. The reference data set includes malware samples from 20 different families and their detailed distribution is given in Table 2. Considering its reliable labeling, the reference set is used to evaluate and fine-tune the empirical parameters for the `MutantX-S`' clustering engine while the large set is used to assess its scalability.

Family	#	Family	#	Family	#
Pilleuz	500	Bredolab	301	Tidserv	59
Koobface	496	Vundo	249	Waledac	34
Silly	489	Almanahe	241	Ackantta	32
Fakeav	489	Sasfis	199	Mebroot	26
Zbot	459	Graybird	166	Hotbar	21
Banker	449	Gammima	126	Qakbot	17
Virut	361	Mabezat	107		

Table 2: Malware families of the reference data set

7.1 Effectiveness of Unpacking Engine

To evaluate the effectiveness of `MutantX-S`' unpacking component, we select a malware program and packed it with 8 popular packers. We then unpack them with `MutantX-S` and compare unpacked files with the original version. Ideally, the unpacked binary should be byte-to-byte identical to the original file. However, this is neither possible (`MutantX-S` does not reconstruct the import table, and the unpacker code is dumped from the memory too), nor necessary for the purpose of malware clustering. As a result, we compared the unpacked files with the original one using two metrics, (i) the difference in their *instruction count* (IC) and (ii) the distance between their N-gram feature vectors (NG), because they are directly related to the clustering accuracy. These results are summarized in Table 3. For most packers, the `MutantX-S`successfully recovered their original binaries with only a 1–6% increase of ICs which is often due to the inclusion of unpacker routines in the dumped memory. Besides, the feature vectors of unpacked binaries are very similar to that of the original binary with most normalized distance measurements below 0.1, where 0 means identical and 1 means completely different. However, `MutantX-S` also failed on certain packers. In particular, the memory dump of Armadillo-packed malware sample still contains a packed version of the binary. A further investigation showed that Armadillo works by unpacking an intermediate executable on disk and creating another process to run this executable [18]. Therefore, memory dumps of an Armadillo-packed file does not contain original instructions. After running `MutantX-S` on the large data set, we have also observed other causes of unsuccessful unpacking, such as malware samples refusing to run in a virtual machine or the time required for unpacking is longer than the threshold used in `MutantX-S` . Despite these rare limitations, the generic unpacking technique used in `MutantX-S` is still effective against popular packers without requiring any specialized unpacking algorithm.

Packer	Diff in IC (%)	NG Distance	Packer	Diff in IC (%)	NG Distance
ASprotect	6.70%	0.133	PEcompact	0.88%	0.068
EXECryptor	3.20%	0.176	UPX	0.88%	0.068
EXEStealth	0.88%	0.071	VMprotect	2.50%	0.1
NSPack	0.87%	0.069	Armadillo		

Table 3: Unpacking effectiveness (IC: Instruction Count; NG: N-gram)

7.2 Malware Clustering Accuracy

We first evaluate and calibrate `MutantX-S` against the reference data set. All of our evaluations were done on a Ubuntu 10.4 machine with Core i7 3.0G CPU and 12GB memory. We use *precision* and *recall* as the main metrics to assess the accuracy of `MutantX-S`' clustering. Suppose that with respect to the original labels (i.e., family names e.g. Table 2), n input malware samples can be grouped into a set of clusters $O = \{O_1, O_2, \ldots, O_o\}$. Assume `MutantX-S` outputs a set of clusters $C = \{C_1, C_2, \ldots, C_c\}$. Then precision P measures how well the individual clusters agree with the original malware classes (i.e., the exactness of clusters), and recall R measures how much the malware classes are scattered across the clusters (i.e., the completeness of each cluster). Formally, we define P and R as:

$$P = \frac{1}{n}\sum_{i=1}^{c} max(|C_i \cap O_1|, |C_i \cap O_2|, \ldots, |C_i \cap O_o|)$$

$$R = \frac{1}{n}\sum_{j=1}^{o} max(|O_j \cap C_1|, |O_j \cap C_2|, \ldots, |O_i \cap C_c|)$$

P will be 1 if all the samples in every cluster C_i are from the same family and R will be 1 if all malware samples from the same family fall into a single cluster (but not necessarily the only family in this cluster). Figure 5 shows the precision and recall of `MutantX-S`' clustering with varying thresholds P_{max} and Min_d (defined in Section 6). The experiment uses 4-gram and the number of hash bits is 12 (i.e., the 4-gram is mapped into 2^{12} hash bins).

From the figure, we observe that `MutantX-S` is able to cluster the samples with the precision ranging from 0.72 to 0.89 (average=0.80). The precision number is smaller than those reported in previous dynamic-behavior approaches, e.g., 0.996 in [22] and 0.984 in [6]. While this difference may be due to different malware sets (and possibly incorrect labeling) used in our experiments, we conjecture that the reason for the higher accuracy of dynamic-behavior approaches is more likely due to its high-level generalization of behavior, with the cost of longer running time and limited coverage (because dynamic analysis can only observe behaviors exhibited during a particular execution. A lot of malware contain triggers in their programs and will not show any malicious behavior unless certain conditions are met e.g, particular date, time, existence of files, control command from the network, etc, These conditions vary across malware samples and are very difficult to simulate during the dynamic anlaysis.) Therefore, `MutantX-S` can provide an alternative way of categorizing malware and is complementary to the behavior-based analysis with better scalability while maintaining reasonably good accuracy. Indeed, Figure 5 shows that it takes only less than 30 seconds to complete the clustering for the entire reference dataset (We also run the K-mean and hierachical clustering on the same dataset and they respectively take 32.3 seconds with precision 0.75 and 51.3 seconds with precision 0.82). In addition, we observe that the recall of `MutantX-S` is around 0.3 and 0.4. However, this low value of recall is not surprising, because we observe that there often exists much diversity across malware variants. For instance, we observe one variant in Vundo family is 10 times bigger in terms of file size than the other Vundo variant. This is possibly due to mislabeled samples, unidentified packers or heavily-obfuscated binaries. Because of the highly diverse variants, `MutantX-S` tends to break

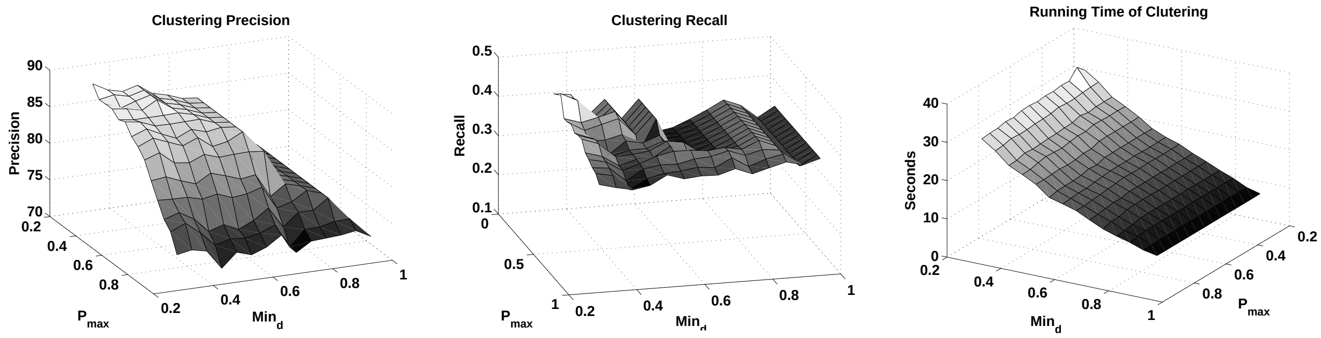


Figure 5: Precision, recall and running time of `MutantX-S`' clustering

the one family into several sub-families, resulting in a low recall value. For instance, `MutantX-S` creates more than 50 clusters for the reference dataset which contains 20 families according to the labels. Albeit less ideal, a breakdown into small subfamilies is acceptable in practice, e.g. predicting labels for unknown samples as we demonstrated later.

Another observation from these results is that P_{max} (the threshold for distances from all data points to their nearest prototypes) has a greater influence on the clustering speed, since a smaller P_{max} forces the algorithm to find more prototypes to cover all the data points, thus requiring more computation. On the other hand, Min_d has a major impact on the clustering accuracy, i.e., increasing Min_d reduces the precision. The reason is that a smaller inter-cluster distance threshold will stop the prototype merging process earlier which will, in turn, reduce the probability of combining unrelated prototypes into a larger cluster. However, the price for this is the overfitting of clustering, i.e., the algorithm tends to create several small clusters. Hence a tradeoff has to be empirically made, as in our later experiments.

7.3 Validity of the Hashing Trick

The main concern in using the hashing trick is the possible loss of information due to the compression of high dimensional features into a lower dimensional space. To evaluate the efficacy of hashing trick, we use different number of hash bins to cluster the reference data set and study its effect on the accuracy. The hash function used in `MutantX-S` is MurmurHash 2.0 [2], a simple hash implementation with uniform value distribution, high throughput, and good collision resistance. As comparison, we also ran the clustering algorithm on the original feature vectors without any hashing trick, which serves as the baseline benchmark and best-possible result achievable without information loss.

Figure 6 compares the precision, clustering time and peak memory requirements with different hash sizes (the number of hash bins ranging from 2^8 to 2^{16} and no hash). Different bars represent the results generated by different parameter combinations. From the left figure, we find that as the hash size increases, the precision improves because the collision probability reduces. In fact, when the hash size is large enough, the probability of collision becomes so negligible that the hashed features vector perform the same as the original ones. For instance, with more than 2^{12} hash bins, the clustering achieves almost the same precision, 0.864, as the original features $P = 0.868$. However, as the hash size becomes smaller, the impact of collision starts to surface. When the number of hash bins reduces to $2^8 = 256$, the precision drops significantly (less than 0.5) for some parameter combinations, due to collision of many critical features (e.g., features indicative of different families are now mapped to the same hash bin) In this regard, a larger hash size is preferable, On the other hand, the middle and right figures in Figures 6 show that a small hash size is very effective in reducing the algorithm's running time and memory footprints. This is because smaller number of hash bins means shorter feature vectors which require less memory for storage and fewer CPU cycles to compute the distance. For instance, as the hash size decreases from 16 bits to 8 bits, the required running time drops from almost 2 minutes to less than 10 seconds and memory requirement from 800 Mbytes to less than 100 Mbytes, at the cost of precision.In practice, a 12-bit hash function is found to be a good compromise, reducing the time and memory requirements by over 80% while still keeping good accuracy[6]. Figure 6 also shows that as the number of malware increases, the hashing trick becomes critical. Without it, the memory requirement could become quickly prohibitively high.

7.4 Impact of N-gram on Performance

Intuitively, as N increases, N-gram becomes more descriptive, because more instructions can be represented by each N-gram, providing better distinguishability However, this comes at the cost of exponential increase in the dimensionality of the resulting feature vectors (m^N where m is total number of different op-

[6]Hence, unless specified otherwise, throughout the paper, the experiments are performed with a 12-bit hash function

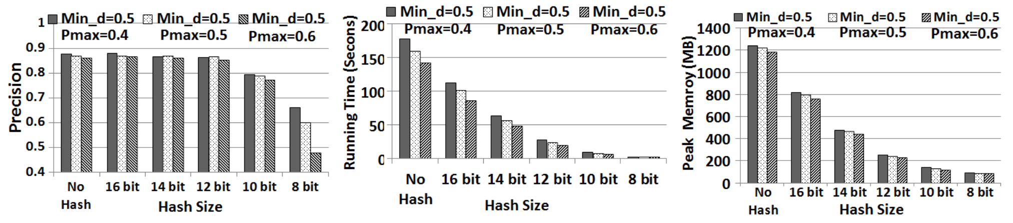


Figure 6: Precision, time, and peak memory with hash bin number ranging from 2^8 to 2^{16} and with no hash trick .

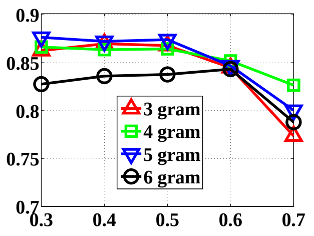


Figure 7: Precision of clustering with different N.

codes), as well as the required storage and computation time. Therefore, previous work that uses N-gram based approaches commonly chose small N (3 or 4). Fortunately, hashing trick enables us to compress the feature vectors and evaluate the performance of large N. Figure 7 summarizes the result with $N = 3, 4, 5, 6$.

From Figure 7, one can observe that use of a larger N value indeed improves the precision, e.g., 4- and 5-grams achieve better precision than 3-gram since larger grams can better capture the underlying instruction semantics. However, the figure also shows that 6-gram performs the worst. This is because the number different 6-grams (i.e., over $6.4 * 10^{12}$) is too large for the 12-bit hash function (4096 hash bins), leading to a large number of collisions between irrelevant features. In `MutantX-S`, we have chosen 4-gram, because the improvement provided by 5-gram is not large enough to warrant the additional storage and computation overheads.

7.5 Scalability of `MutantX-S`

In this subsection, we evaluate the scalability and accuracy of `MutantX-S` on the large malware data set with over 130,000 samples. We ran `MutantX-S` on the entire set with different parameters and plotted the results in Figure 8. The right figure in Figure 8 shows the amount of time for clustering the entire set. the value P_{max} seems to have a more significant impact on the running time. For example, when P_{max} is set to 0.5, the clustering takes less than 1 hour which is almost half of the time when P_{max} is set to 0.2. As explained before, P_{max} determines the number of prototypes extracted from the input data which determines the total number of distance computations required for clustering. Although larger P_{max} leads to shorter running time, the left figure in figure 8 illustrate the correlation between a large P_{max} and the reduced clustering precision, i.e. increasing P_{max} from 0.2 to 0.5 reduces the precision by almost 10%. This can be explained as follows: a large P_{max} allows each prototype to cover a large portion of the space, thus increasing the possibility of including samples from irrelevant families. With a reasonable setting (e.g., $Min_d = 0.5$ and $P_max = 0.4$), `MutantX-S` is able to complete the clustering of over 130 K malware in less than 1.5 hours with the precision close to 0.82.[7] The peak memory usage is around 3.6 GB. These results indicate that `MutantX-S` is very efficient in handling a large number of samples and thus has the potential to keep up with the huge influx of malware variants received nowadays.

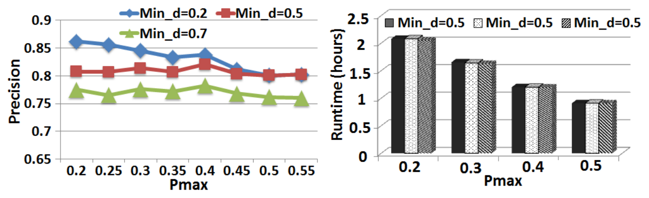


Figure 8: Precision and running time of `MutantX-S`' clustering over 130 K samples

7.6 Predicting Labels of Unknown Malware Samples

So far, we have evaluated `MutantX-S` using the data set of known malware families. In a realistic scenario, e.g., in AV companies, `MutantX-S` is more likely to be used to analyze new incoming malware and predict their family labels. In such a scenario, incoming malware are analyzed and labeled according to their association with the closest kin in the previously-analyzed samples. To simulate this situation, we need a chronological order of malware samples according to their creation time. We extract the creation time for each malware from their IMAGE_FILE_HEADER. IMAGE_FILE_HEADER is a standard header in the PE file and contains a timestamp field that is set by the compiler at the compilation time. We use this timestamp to bucket malware programs into months and select one year worth of malware (more than 40 thousand unique samples). Figure 9 shows the distribution of the number of new malware samples across all months. Next, we use these malware to simulate the process of determining the labels for new incoming samples.

[7]The recall for the large data set is around 0.25 because of breaking the samples from large malware families into relatively small groups.

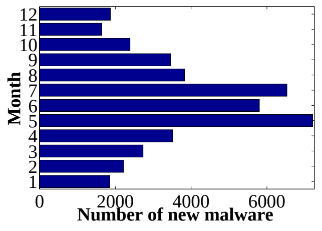


Figure 9: Number of new samples in each month used to evaluate prediction capability

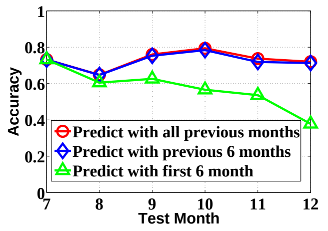


Figure 10: Accuracy of applying `MutantX-S` to predict family labels for unknown malware

Specifically, we separate the malware program into *training set* and *testing set* based on their creation time in order to simulate the scenario where AV companies have analyzed the malware from training set and try to predict the labels for newly received malware (testing set). The test set consists of malware samples from each month between July and December (these months are "test months"). Then, we choose 3 different training set. For the first training set, we use samples from all months from January to one month before the test month. For instance, if the test month is September, the training months are January through August. For the second training set, we use 6 months prior to the test month, i.e., if September is the test month, March through August will be the training months. Finally, as a controlled experiment, we keep the the first 6 months (i.e., January through June) as the training month regardless of test months. Given any training set, `MutantX-S` create a set of clusters C_i $(i = 0, 1, ...n)$. Each cluster has a label $L(C_i)$ determined by the majority family labels of the constituent malware samples. Then `MutantX-S` determines the family label $L(x_j)$ of the new sample x_j in the test month based on the label of the cluster that is closest to x_j i.e. $L(x_j) = L(C_i)$ where $d(x_j, C_i) = \min(d(x_j, C_k)) \forall k = 0, 1, ..n$. We then compare this predicted family label with the original label of x_j, and plot the percentage of correctly predicted samples in Figure 10. The first observation from the figure is that malware are constantly evolving and information obtained from previous clustering can become obsolete quickly, as shown by the bottom green line where we kept on using the same first 6 months as the training data and the prediction accuracy degraded rapidly from 0.7 in July to below 0.4 in December. In contrast, if we use the full history as the training data, the accuracy stays consistently in the 0.7 − 0.8 range (the top red line in Figure 10), thanks to the up-to-date information from the recent malware. However, in reality, due to the resource constraints (e.g., storage), it may not be possible to keep the entire set of previous malware samples. It is more efficient to use only the most recent history, e.g., 6 months as in the middle blue line. From Figure 10, one can see that the result is very close to that of using the full history, with only small decreasing about 2 to 3%. These results imply that there exists strong temporal correlation among malware variants which can be exploited by `MutantX-S` in predicting the labels for unknown malware samples.

8. LIMITATIONS AND IMPROVEMENTS

We now discuss limitations of the current `MutantX-S` prototype that could be exploited by adversaries to degrade its clustering effectiveness. As a static-analysis approach, `MutantX-S` is vulnerable to binary/instruction-level obfuscation. First, even with the generic unpacking algorithm, `MutantX-S`is less effective against advanced packers, such as Armadillo that employ sophisticated protection mechanisms, e.g., driver-level protection, anti-debug, anti-emulation, etc. Specialized unpacking tools have been developed for these packers and they can be incorporated into `MutantX-S`to combat these packers. Second, `MutantX-S` extracts features from disassembled malware code. Unfortunately, producing correct disassembly is often very challenging and many anti-disassembly tricks[29] can be used to confuse a disassembler such as mixture of code and data, making an infeasible conditional jump to the middle of next instruction, etc. Although the current `MutantX-S` prototype does not handle these types of obfuscation for simplicity, there are a variety of techniques [15] proposed to mitigate these problems. Third, `MutantX-S` relies on the similarity of code instructions to cluster malware samples. It is possible to create syntactically distinct but semantically similar variants through heavy instruction-level obfuscation. To address these problems, `MutantX-S` could incorporate more advanced deobfuscation techniques [25, 20] and normalize the malware codes before clustering them. Note that dynamic-behavior-based approaches do not suffer from this limitation, but they come with their own deficiencies—limited coverage, scalability and specific evasion techniques too. Therefore, the goal of `MutantX-S` is not to replace the dynamic approaches, but to complement them and mitigate their weaknesses. Finally, `MutantX-S` cannot handle file infector or parasitic malware types which inject themselves into host executables. This is a limitation for any similarity based clustering, regardless static and dynamic, because most features are from the host executables rather than the virus. Such parasitic malware is a matter of our future inquiry.

9. CONCLUSION

In this paper, we have presented the design, implementation and evaluation of a malware clustering system based on static features, called `MutantX-S`. `MutantX-S`can accurately and efficiently group malware variants according to the similarity in their code instructions. `MutantX-S` converts each malware program into a compact but effective opcode representation and performs prototype-based clustering on the corresponding N-gram feature vectors. `MutantX-S` incorporates a generic unpacking technique to maximize the capability of analyzing the malware's original instructions. To ensure the scalability of clustering, `MutantX-S` uses a combination of a hashing kernel that reduces the dimensionality of feature vectors and a close-to-linear time prototype-based clustering that uses a small set of representative samples for fast data organization. Equipped with these techniques, `MutantX-S` is experimentally shown to be able to process more than 100,000 malware samples within a few hours. As a static-analysis approach, `MutantX-S` is expected to be very effective and can be combined with existing dynamic-behavior-based system to provide the level of accuracy and coverage required to pace with the current malware sample submission rate.

10. REFERENCES

[1] Peid 0.95. http://www.peid.info/, 2008.

[2] Appleby, A. Murmurhash 2.0. http://sites.google.com/site/murmurhash/.

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

[4] ASpack Software. Aspack. http://www.aspack.com/.

[5] Bailey, M., Andersen, J., Mao, Z. M., and Jahanian, F. Automated classification and analysis of internet malware. Tech. rep., In Proceedings of Recent Advances in Intrusion Detection (RAID07, 2007.

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

[7] Christodorescu, M., and Jha, S. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium* (2003), pp. 169–186.

[8] Gonzalez, T. Clustering to minimize the maximum intercluster distance. In *Theoretical Computer Science* (1985), vol. 38, pp. 293–306.

[9] Guo, F., Ferrie, P., and Chiueh, T.-C. A study of the packer problem and its solutions. In *11th International Symposium on Recent Advances in Intrusion Detection: RAID'08* (2008), pp. 98–115.

[10] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer-Verlag, 2009.

[11] Jang, J., Brumley, D., and Venkataraman, S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of CCS'11* (2011).

[12] Karim, M. E., Walenstein, A., Lakhotia, A., and Parida, L. Malware phylogeny generation using permutations of code. *JOURNAL IN COMPUTER VIROLOGY 1* (2005), 13–23.

[13] KilianWeinberger, Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. Feature hashing for large scale multitask learning. In *Proceedings of the 26 th International Conference on Machine Learning* (2009).

[14] Kolter, J. Z., and Maloof, M. A. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research 7* (2006), 2006.

[15] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 18–18.

[16] Lee, T., and J.Mody, J. An automated virus classification system. In *Proceedings of VIRUS BULLETIN CONFERENCE OCTOBER 2005* (2005).

[17] Manning, C. D., Raghavan, P., and Schutze, H. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[18] Martignoni, L., Christodorescu, M., and Jha, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *In Proceedings of the Annual Computer Security Applications Conference (ACSAC* (2007).

[19] Pontello, M. Trid. http://mark0.net/soft-trid-e.html.

[20] Raber, J., and Laspe, E. Deobfuscator: An automated approach to the identification and removal of code obfuscation. *Reverse Engineering, Working Conference on 0* (2007), 275–276.

[21] Rieck, K., Holz, T., Willems, C., Düssel, P., and Laskov, P. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, pp. 108–125.

[22] Rieck, K., Trinius, P., Willems, C., and Holz, T. Automatic analysis of malware behavior using machine learning. tech report, Berlin Institute of Technology, 2009.

[23] Shawe-Taylor, J., and Cristianini., N. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

[24] Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., Strehl, A., and Vishwanathan, V. Hash kernels. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)* (2009).

[25] Udupa, S. K., Debray, S. K., and Madou, M. Deobfuscation: Reverse engineering obfuscated code. *Reverse Engineering, Working Conference on 0* (2005), 45–54.

[26] UPX. the ultimate packer for executables. http://upx.sourceforge.net/.

[27] VxHeaven. Vxheaven virus collection. http://vx.netlux.org/, 2010.

[28] Wicherski, G. pehash: A novel approach to fast malware clustering. In *2nd Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET'09)* (2009).

[29] Yason, M. V. The art of unpacking. https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf, 2007.