Similarity-based matching meets Malware Diversity

Mathias Payer[1]*, Stephen Crane*[2]*, Per Larsen*[2]*, Stefan Brunthaler*[2]*, Richard Wartell*[3]*, Michael Franz*[2]
[1] *UC Berkeley,* [2] *UC Irvine,* [3] *Mandiant Corporation*

Abstract

Similarity metrics, e.g., signatures as used by anti-virus products, are the dominant technique to detect if a given binary is malware. The underlying assumption of this approach is that all instances of a malware (or even malware family) will be similar to each other.

Software diversification is a probabilistic technique that uses code and data randomization and expressiveness in the target instruction set to generate large amounts of functionally equivalent but different binaries. *Malware diversity* builds on software diversity and ensures that any two diversified instances of the same malware have low similarity (according to a set of similarity metrics). An LLVM-based prototype implementation diversifies both code and data of binaries and our evaluation shows that signatures based on similarity only match one or few instances in a pool of diversified binaries generated from the same source code.

1 Introduction

The malware (malicious software) landscape is constantly evolving. There are no longer tens of thousands of different malware threats that are currently active but only few different malware families that often share common source code. Current malware detection engines (malware scanners and anti-virus engines) use a combination of signatures, partial matching, regular expressions, and heuristics to classify binaries as either malicious or benign. Malware therefore faces a detection problem on current systems due to shared source code and a low number of currently active malware families.

Current malware addresses the detection problem using packers [41, 51] (small pieces of code that obfuscate the actual malware code from analysis), semi-automatically generating new binaries every couple of hours. Even Symantec, one of the top anti-virus companies, declares that signature-based similarity metrics are no longer effective against top threats [12]. Packers usually work on binaries, however, and are therefore limited in the expressiveness of the changes due to missing high level information.

Software diversity [7,9,11,16,17,20–22,28,35,43] on the other hand uses a compiler to produce functionally equivalent binaries that differ substantially at the implementation level. Software diversity provides protection with quantifiable probability from software exploits that rely on a known data or code layout. Software diversity is also used to thwart reverse-engineering and tampering; up until this point, software diversity has been a defensive capability.

Malware diversity [44] tailors software diversification to the needs of malware authors. Malware diversification ensures that two diversified binaries share neither large amounts of data nor common instruction sequences. Regular software diversification (i) typically diversifies code regions while data regions remain constant, (ii) avoids performance degrading changes, and (iii) is deterministic, i.e., the diversification is reproducible. Malware diversification shifts these design decisions: the diversification engine diversifies both code and data (otherwise signatures could match data), maximizes the (byte-wise and structural) differences, and minimizes the largest common subsequence of shared code/data. Reproducibility is only helpful to report and troubleshoot bugs in deployed software; this is not of concern to malware authors who can only reliably test malware prior to launching it.

Obfuscation [9] is closely related to software diversity as both techniques rely on randomizing transformations. Obfuscating transformations protect binaries against reverse engineering by hiding the implemented algorithm. Malware diversity is complementary to obfuscation as it modifies the computation of every single instance but debugging and reverse engineering of individual instances are not affected.

Our malware diversification engine extends the

LLVM-based multicompiler [22, 44] and changes the code of the application by replacing instructions, instruction reordering, garbage insertion, and several types of control-flow randomization like reordering basic blocks. To change the data of an application, the malware diversification engine uses different data encodings.

The contributions of this paper are as follows:

1. a description of malware diversity, a technique that extends software diversification to create malware instances with low similarity to each other;

2. a detailed evaluation of a prototype implementation demonstrating the effectiveness of malware diversification along several similarity metrics.

2 Background and related work

Malware diversity extends software diversity by combining code and data diversity (to obfuscate data regions along with the code regions). Malware diversification is effective when malware detection mechanisms fail to identify two diversified binaries using the same signature. Put differently, we can measure the effectiveness by finding common features that are present in both diversified malware samples (according to some similarity metric).

2.1 Malware detection and evasion

Current malware scanners combine different matching techniques to detect malicious code. Most systems use a combination of different matching techniques like hash based matching (using a cryptographic hash for the entire binary or individual sections of the binary), sequence based matching (if two binaries share a common sequence they are considered equal), expression based matching (if the regular expression matches both binaries they are considered equal), and heuristics based matching (if a binary matches a given heuristic it is considered malicious). This list is based on ClamAV [29], a well-known open-source malware scanner; other scanners rely on similar techniques. According to Huang and Tsai [23] the average matched pattern length is longer than 25 bytes and only 43 out of more than 83,000 signatures are shorter than 10 bytes. The likelihood of false positives decreases with increased matched pattern length.

Several new approaches for both malware detection and malware matching have been proposed to address the limitations of the existing signature-based techniques. Approaches like malware normalization [6] normalize a binary to a common form but are limited to predefined obfuscation patterns and cannot undo high-level transformations like register reallocation.

Heuristics based malware detection tries to match the behavior of a binary to a specific sequence of actions (such as system or library calls) when executed in a sandbox. Malware uses subtle differences between the sandbox and a real system [5, 18] to detect virtualized platforms [14, 42, 46, 48] and stops execution. Approaches that detect sandbox evasion [2, 14, 24, 26, 30, 34, 36] are useful tools for analysts but usually too heavyweight to be used on a consumer's machine.

2.2 Packers and binary polymorphism

A packer [25, 37, 39, 45, 47] (or crypter) is an application that obfuscates a malicious application with the intention to hide it from malware scanners or to make debugging and reverse engineering harder. Packers are historically based on encryption but moved to oligomoprhic, polymorphic, and metamorphic transformations [40, 53]. Botnet operators can strengthen polymorphic transformations by randomizing them on a per-machine basis, for example by using `perl` scripts to use non-standard transformation algorithms [1].

Since packers are complex to construct, many malware authors reuse existing solutions. Attackers are increasingly shifting to less common packers, customized packers, and obfuscating packers [4]. However, anti-virus scanners can still detect packed binaries due to their special characteristics. These include "weird" section names, sections with high Shannon's entropy due to compression, few imported functions, and unusual entry code [45, 54]. Finally, many anti-virus scanners can even unpack and scan the payloads of known packers [25, 37, 47], allowing the use of previously discussed detection techniques.

2.3 Software diversification

Software diversification [7, 9, 11, 16, 17, 20–22, 28, 35, 43] is a promising technique. Diversity can be used to (i) increase the resilience of software against attacks [7, 11, 16, 17, 19–22, 28, 35, 43], to (ii) hide steganographic messages in binaries [13], and to (iii) protect software against tampering [8]. Software diversification constructs functionally equivalent programs that differ in their code and/or data layout. A diversification engine uses several (compiler) techniques to randomize the code and data comprising an application: (i) instruction replacement and reordering, (ii) variable substitution, (iii) register reordering, (iv) control flow changes, (v) adding side-effect free instructions, (vi) instruction set randomization [3, 27, 50, 55], (vii) instruction stitching [38], or (viii) covert computation [49]. Larsen et al. [32] survey the area of automated software diversity in greater detail.

3 Malware diversification

Malware diversification [44] is a form of software diversification that focuses on avoiding similarity-based detection by malware scanners. To this end, malware diversity modifies code and data regions of binaries. As a result, malware analysts are unable to generate a signature that matches more than few instances of a malware, if any two binaries only share few instructions at the same offsets, share no common data, and have dissimilar control flow graphs. A notable difference to other software diversification techniques is that static data must be diversified alongside code and static data (otherwise a signature would just match the static data).

Code diversification for malware builds on existing software diversification mechanisms like instruction replacement, instruction reordering, register reordering, and changing control flow by splitting and reordering basic blocks, inlining, outlining, and adding opaque predicates. Malware diversification configures software diversification to maximize diversity in the generated code and to minimize similarity between multiple diversified binaries.

Data diversification changes the encoding of static data in the binary. Due to limited knowledge of the structure of data, malware diversity resorts to a form of obfuscation [9] to hide the actual static data. Our malware diversification engine uses a simple encoding scheme that is applied to static data during compilation. All static data is encoded with a random key and simple arithmetic operations (e.g., `xor`) decode the data at runtime. The decoding function is diversified along with all other code.

The data diversification presented here is only a simple technique that can be strengthened, e.g., by encrypting the data. Also, most programs have little static data compared to the amount of code. Note that malware diversification does not result in binaries with the same special properties that makes it easy for anti-virus software to detect packers (cf. Section 2.2)—rather, the resulting binaries look like variations of benign programs.

We implemented a simple malware diversification engine on top of the existing multicompiler [22, 44] that extends the LLVM [33] compilation framework version 3.4. The compiler is organized as a sequence of passes which transform the instruction stream. By modifying the existing compiler transformations and adding new ones, we enable malware diversification. Our prototype currently does not use runtime data structure diversification [35].

Unlike traditional compiler optimizations that choose whether to transform the code or not based on program analysis, heuristics, and profile feedback, our diversifying transformations use a random number generator to "flip a coin" at every opportunity to diversify. We perform the following forms of diversification: (i) instruction replacement, randomly swapping `mov` and `lea` instructions, (ii) instruction reordering, (iii) register reordering, (iv) `nop` and garbage instruction insertion, (v) control flow randomization and randomizing the layout of basic blocks, and (vi) static data obfuscation. The resulting compiled binary is stripped to remove all symbol names (variable names and function names) in the final diversified binary.

Not only are there several other techniques that we could add to our diversification engine (cf. Section 2.3), the diversity generated by each of the existing passes could also be increased. Consequently, we believe that malware writers will not find it difficult to replicate our approach. The current implementation targets the IA32 ISA but the concept is portable to any instruction set and operating system combination.

4 Evaluation

This section evaluates the prototype implementation of our malware diversification engine. Using our prototype implementation and a set of programs we produce 10 diversified instances for each binary and evaluate the similarity between different binaries according to a set of metrics. Unfortunately, a lot of Windows malware is compiled using Visual Studio using the Microsoft C/C++ compiler. These sources are often not compatible to Clang/LLVM due to Microsoft specific intrinsics in system C++ headers. All benchmarks are executed on an Intel Core i3-3770 CPU with 16GB RAM on Debian 7.1 using the Linux 3.6.1 kernel.

We use the following applications: all C/C++ applications of the SPEC CPU2006 benchmarks, kbackdoor (a simple Windows malware), a simple port scanner, pwdump (a Windows password recovery tool for LM and NTLM hashes), and mimikatz (a Windows password recovery tool that targets lsass.exe). We successfully compiled these programs using our diversifying compiler. Some of these programs are very small: the simple port scanner and `mimikatz` are below 10KB, `pwdump` and `kbackdoor` are below 75KB. Only nmap can be considered a "large" binary with 3.4MB. The small size makes is harder for the diversifying compiler to produce a highly diverse population of binaries.

4.1 Common subsequences

Malware matching relies on signatures: common subsequences that classify the malware uniquely (see Section 2.1). Common subsequences that are present in many (or all) diversified instances are candidates for signatures. We search all instances for common subsequences longer than 10 bytes, resulting in an over-

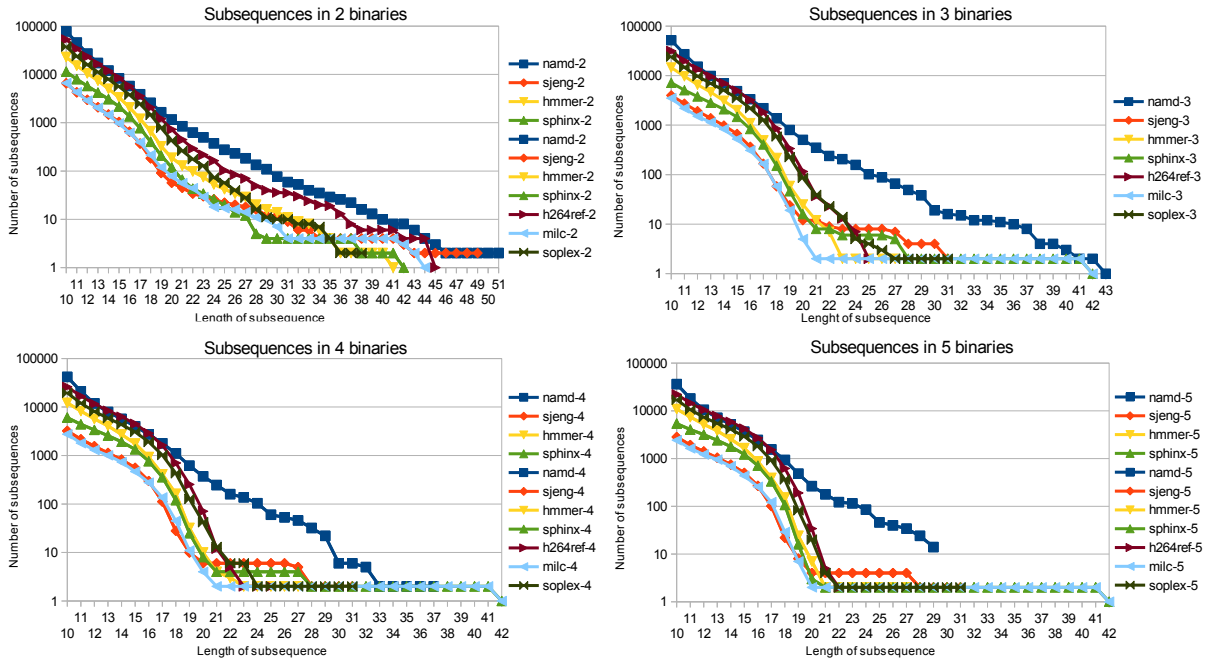

Figure 1: List of common subsequences for SPEC CPU2006 benchmarks (log scale).

approximation of all possible signatures (e.g., duplicate code, register spill code, instruction sequences that call library routines with common parameters, or function prologues and epilogues result in many shared substrings but are often not usable as signatures).

As indicated by Huang and Tsai [23] the average signature is longer than 25 bytes; shorter signatures are not unique to the malware and lead to a high number of false positives. Figure 1 shows common substrings for diversified versions of SPEC benchmarks shared among a set of 2, 3, 4, or 5 diversified versions. The figure highlights two interesting results: (i) the number of shared subsequences of a given length for a benchmark is lower the more diversified versions are compared (i.e., there are less shared substrings between three diversified binaries than between two) and (ii) the number of shared substrings drops logarithmically with increasing length of the substrings.

The comparison shows that most common sequences are between 10 bytes (10 bytes is the cutoff length) and 20 bytes of length. Most of these short sequences are function epilogues and `nop` sleds to align the next function to a 16 byte address or `nop` chains before a function prologue. The number of common subsequences drops drastically with increasing length; only very few subsequences are longer than 30 bytes. This comparison shows the effectiveness of diversification to counter common subsequences in binaries: two different benchmarks can have higher similarity than three diversified binaries.

We manually looked through the identified subsequences and classified them into one of the following categories: (i) 10 to 15 byte `nop` sleds to, e.g., align functions to 16 byte offsets, (ii) function call sequences, pushing static arguments or arguments at specific stack offsets, (iii) `mov` sequences that load/store memory into registers, e.g., to initialize structures, (iv) static start code added by the compiler (e.g., the function that executes before main is called), and (v) potential signatures. We found that there are only few potential signature candidates and all of them use registers where register reordering will introduce diversity for larger sets of binaries.

4.2 Instruction frequencies and n-grams

This similarity metric groups either individual instructions or instruction mnemonics of a binary, removing register information and memory access information (e.g., `mov %eax, %ebx` and `mov %ecx, %eax` share the same instruction mnemonic). This histogram can be used as a fingerprint of the malware. We define the similarity measure S between two binaries bin_1 and bin_2 as follows:

$$\texttt{freq}(\text{mnem}, B) = \frac{\text{mnem}_{\text{total}}(B)}{\text{instrs}_{\text{total}}(B)}$$

$$S = 1 - \sum_{\forall i \in \text{instr}} \frac{|\texttt{freq}(i, \text{bin}_1) - \texttt{freq}(i, \text{bin}_2)|^2}{2}$$

The frequency of one instruction (or mnemonic) is the number of times this instruction is used in the binary divided by the total number of instructions. A table of frequencies for each instruction is the histogram of a binary. The similarity between two binaries is defined as the sum of all absolute squared differences between each mnemonic frequencies. Similarity is a natural number between 1 (every mnemonic occurs an equal number of times in both binaries) and 0 (the two binaries share no instruction mnemonics).

In our experiments with the SPEC CPU2006 benchmarks we found that such simple fingerprints (both instruction and mnemonic based similarity) are not significant enough to distinguish diversified binaries of the same program from other programs with high confidence. For many benchmarks the similarity between two different benchmarks is as high as the similarity between two diversified versions of the same benchmark. An interesting observation of this simple fingerprinting experiment is that different programs with different functionality have very high similarity. Only few instructions differ overall.

A straight-forward extension of instruction frequencies are n-gram frequencies where n instructions are bundled together into one class (a 2-gram instruction sequence would be, e.g., a mov followed by a pop). The simple instruction frequency defined above represents the 1-gram frequency. We evaluated the n-gram frequencies for the SPEC benchmarks for $n \in \{2,3,4,5\}$ and found similar results to $n = 1$: n-gram similarity is not significant enough to match diversified versions of the same binary. Actually, n-grams offer slightly lower similarity between diversified versions for $n > 1$ than for $n = 1$.

4.3 Jaccard similarity

Following the results from the naive malware fingerprinting in the previous section we refine our similarity metric and use the Jaccard Similarity (JS) coefficient to compare two binaries.

JS is a statistical metric used to compare the similarity and diversity of two sample sets by dividing the size of the intersection of the two sets with the size of the union of the two sets:

$$JS(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

For each binary we construct a set of instruction frequencies, i.e., we count for each instruction type how many times it is used in the binary and we then calculate the JS based on these two sets.

The JS effectively highlights differences between sets in terms of instruction frequences or instuction types. Two binaries that are exactly the same have a JS of 1 while binaries that share no mnemonics with the same count of instructions have a JS of 0.

	kbackdoor	mimikatz	pwdump	nmap	sps
kbackdoor	**0.68**	0.27	0.29	0.43	0.28
mimikatz		**0.68**	0.29	0.28	0.3
pwdump			**0.8**	0.27	0.28
nmap				**0.71**	0.28
sps					**0.77**

Table 2: Jaccard similarity for our malware set with themselves and with each other. Higher values indicate higher similarity.

Table 1 shows the JS coefficient measurements of diversified versions of all C/C++ SPEC CPU2006 benchmarks. Each benchmark is diversified 3 times. If the benchmark is compared with itself then we report the average JS of all three diversified binaries between each other (e.g., for bin1, bin2, bin3 we report the average of bin1-bin2, bin2-bin3, and bin1-bin3). If two different benchmarks are compared then we report the average of all 9 individual JS between each diversified version of the first and second benchmark. JS is somewhat effective: some diversified SPEC benchmarks have a higher similarity with diversified copies of themselves compared to other diversified benchmarks. Some benchmarks (lbm, libquantum, and milc) have similarity with themselves of almost 0.5 or higher. These benchmarks are identifiable due to specific floating point instructions. Other benchmarks that mostly execute integer instructions are hard to identify as the similarity between different diversified binaries is close to (or even lower) than other benchmarks.

Table 2 shows the JS similarity for our set of malware programs. Due to the small size of these programs and the single purpose the JS similarity can be used to successfully identify diversified versions. For the two larger programs (kbackdoor and nmap) the difference between the JS similarity for diversified binaries and the JS similarity for different binaries is smaller than for the other benchmarks due to the additional functionality in these programs.

Overall we can conclude that JS is effective in identifying diversified binaries of some (smaller) benchmarks with particular instruction sequences. On the other hand JS cannot be used as a general approach to identify any diversified binary or more complex program. Especially as the sample set of benign binaries grows larger it will be hard to get enough confidence to identify diversified versions of a piece of malware relative to all the benign fingerprints.

As a potential countermeasure malware diversity can decrease the JS by adding additional garbage instructions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Xalan (1)	**0.81**	0.34	0.13	0.25	0.34	0.11	0.33	0.12	0.13	0.13	0.12	0.11	0.33	0.34	0.33	0.33	0.13	0.33	0.11
astar (2)		**1.00**	0.15	0.24	0.34	0.12	0.33	0.11	0.13	0.13	0.14	0.13	0.32	0.34	0.32	0.33	0.13	0.33	0.12
bzip2 (3)			**0.38**	0.24	0.34	0.11	0.34	0.12	0.14	0.15	0.14	0.12	0.32	0.34	0.33	0.33	0.14	0.34	0.12
dealII (4)				**0.62**	0.33	0.11	0.33	0.11	0.13	0.13	0.12	0.11	0.32	0.34	0.33	0.34	0.13	0.34	0.12
gcc (5)					**1.00**	0.12	0.33	0.11	0.14	0.13	0.12	0.12	0.32	0.33	0.33	0.33	0.13	0.34	0.11
gobmk (6)						0.33	**0.34**	0.12	0.13	0.14	0.12	0.11	0.32	0.33	0.33	**0.34**	0.13	**0.34**	0.12
h264ref (7)							**1.00**	0.12	0.13	0.13	0.12	0.11	0.32	0.35	0.35	0.34	0.12	0.33	0.12
hmmer (8)								0.33	0.13	0.13	0.12	0.12	0.32	**0.34**	0.32	**0.34**	0.13	**0.34**	0.12
lbm (9)									**0.39**	0.13	0.13	0.11	0.32	0.33	0.33	0.33	0.14	0.33	0.12
libquantum (10)										**0.38**	0.13	0.12	0.32	0.34	0.33	0.33	0.14	0.33	0.12
mcf (11)											**0.36**	0.12	0.32	0.33	0.32	0.33	0.13	0.33	0.12
milc (12)												**0.33**	0.33	0.33	0.32	0.33	0.13	0.33	0.12
namd (13)													**0.96**	0.34	0.32	0.33	0.13	0.34	0.11
omnetpp (14)														**0.99**	0.34	0.33	0.14	0.33	0.12
perlbench (15)															**0.97**	0.33	0.12	0.33	0.12
porvray (16)																**1.00**	0.13	0.34	0.12
sjeng (17)																	**0.37**	0.33	0.12
soplex (18)																		**0.99**	0.12
sphinx (19)																			**0.34**

Table 1: Jaccard similarity for all C/C++ SPEC benchmarks with themselves and with each other.

alongside the diversified instructions. Attackers can deliberately choose the type and number of garbage instructions to minimize the JS score as described by De Sutter et al. [52]. Such a scheme distorts the fingerprint of a given binary and lowers the utility of the JS metric. The implementation of this additional distortion tactic is left as future work.

4.4 Graph based similarity

Bindiff is a plugin for the Interactive DisAssembler (IDA) that compares two binaries and evaluates their structural similarity based on a set of graph based matching techniques. Bindiff works by recovering and comparing approximations of the actual control-flow and call graphs of two diversified binaries. The core algorithm is described in the original bindiff paper [15]. Since bindiff is a commercial tool, now developed by Google, we expect substantial improvements have been made in the interim.

Due to the graph-based comparision approach, bindiff is unaffected by those of our diversification techniques that leave the flow of control unaffected.

Table 3 shows the similarity of a subset of the SPEC benchmarks and our malware set. When comparing diversified versions of the same program we use a set of 5 diversified versions and report the lowest similarity in this set. In our tests, bindiff showed similar similarity across diversified versions.

In general, bindiff achieves a higher similarity than the Jaccard similarity due to the combination of multiple different matching algorithms (including some graph-based matching). On the other hand even bindiff cannot detect very high similarity (the maximum similarity is 53.8%). Bindiff-like similarity metrics can be used to defeat diversity but at the price of additional manual analysis; it takes several minutes to analyze a pair of binaries by hand using IDA Pro and bindiff, resulting in the reported low similarity numbers.

We identified function calls to system libraries (e.g. `libc`) as a major source of structural information that assisted bindiff in computing similarity. We plan to substantially extend our set of transformations that add structural diversity to binaries. In particular, our experiments indicate that calling library functions through randomly generated wrapper functions substantially affects similarities reported by bindiff. Coppens et al. [10] studied structural transformations that defeat bindiff to protect software patches against reverse engineering; we can benefit from this catalog. An additional, orthogonal diversification technique weaves a benign application into the diversified malware and interleaves the instruction stream of both programs using unused resources in the malicious program to execute additional superfluous computation.

5 Discussion

Diversity reduces the similarity between different instances of a binary enough to disable direct, similarity-based matching. Malware diversification can use existing degrees of freedom in the compilation process and

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
astar (1)	0.432	0.118	0.064	0.067	0.052	0.085	0.109	0.085	0.128	0.051	0.109	0.040	0.104	0.086
bzip2 (2)		0.310	0.046	0.050	0.042	0.112	0.099	0.085	0.095	0.048	0.084	0.049	0.107	0.074
h264ref (3)			0.363	0.109	0.012	0.026	0.048	0.023	0.089	0.018	0.055	0.014	0.046	0.105
hmmer (4)				0.396	0.010	0.027	0.048	0.025	0.093	0.020	0.056	0.023	0.052	0.126
kbackdoor (5)					0.377	0.076	0.065	0.090	0.028	0.311	0.037	0.359	0.037	0.025
lbm (6)						0.440	0.115	0.244	0.062	0.139	0.090	0.107	0.098	0.037
libquantum (7)							0.394	0.103	0.105	0.070	0.108	0.067	0.099	0.083
mcf (8)								0.331	0.056	0.130	0.071	0.141	0.080	0.035
milc (9)									0.381	0.034	0.109	0.033	0.103	0.110
mimikatz (10)										0.502	0.060	0.413	0.048	0.027
namd (11)											0.538	0.043	0.101	0.074
pwdump (12)												0.482	0.046	0.022
sjeng (13)													0.343	0.073
sphinx (14)														0.402

Table 3: Similarity of diversified versions according to bindiff.

the resulting binaries to adapt to newly proposed counter measures and use additional diversification (or garbage insertion). This situation will result in yet another security arms race between attackers and defenders until either the diversity in the compilation process is exhausted (which is unlikely) or malware diversity will circumvent all detection mechanisms.

A possible countermeasure is a canonicalization of diversified binaries that undoes individual diversifications. In case of `nop` insertion, we recognize that stripping all `nop`s from a binary[1] does not produce the original binary before diversification. The reason being that we cannot distinguish between `nop`s inserted during diversifications and `nop`s inserted for other purposes (e.g., alignment). However, stripping out `nop`s leaves us with a "canonical" version of the binary in the sense that all similar binaries diversified with `nop` insertion result in the same canonical binary. A similar argument holds for instruction replacement, instruction reordering and register assignment reordering. In case of instruction reordering, we can canonicalize the binary by computing instruction histograms (using only instruction mnemonics in the presence of register assignment reordering) and then match up basic blocks using the control flow graph.

In general, we note that the relationship between diversified and canonicalized binaries is not bijective; that two diversified binaries map to the same canonical representation makes it likely, but not certain, that they share the same source code. Furthermore, reordering more code features (registers, instructions, basic blocks, functions) increases the likelihood of two, distinct programs having the same canonical representation. Even though this makes false positives a potential concern, we expect that a diversification-aware matching strategy based on canonicalization is more accurate than any diversification-oblivious attempt at classifying binaries. On the other hand, randomly inserting instruction sequences from benign applications reduces our ability to compute a canonical version of binaries. We plan to evaluate this type of defense as future work.

[1] Our discussion relies on disassembly of malware which often employs anti-disassembly techniques. We consider this an orthogonal concern and refer to the solutions in the literature (e.g. [31]).

6 Conclusion

Malware detection engines rely on effective and efficient similarity metrics to classify binaries as malicious or benign. Malware diversity uses software diversity to break this assumption and randomly diversifies both code and data of programs. Similarity-based metrics are no longer effective due to the variations in the binary layout; our experiments confirm that malware diversity results in very short common subsequences and breaks other similarity metrics as well. The structural metrics used by bindiff, on the other hand, are not efficient to compute.

Malware diversity enables a new class of malware that generates a virtually unlimited number of unique malware instances. Our experiments and the discussion of protective measurments show that malware diversity is powerful enough to counter new detection mechanisms by exploiting additional opportunities for diversification.

References

[1] ANONYMOUS. Iama a malware coder and botnet operator, ama. `http://www.reddit.com/r/IAmA/comments/sq7cy/iama_a_malware_coder_and_botnet_operator_ama/`, May 2012.

[2] BALZAROTTI, D., COVA, M., KARLBERGER, C., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Efficient detection of split personalities in malware. In *NDSS'10: Proc. Network and Distributed System Security Symp.* (2010).

[3] BARRANTES, E. G., ACKLEY, D. H., FORREST, S., AND STEFANOVI, D. Randomized instruction set emulation. *ACM Transactions on Information System Security 8* (2005), 3–40.

[4] BUSTAMANTE, P. Packer (r)evolution, 2008. `http://research.pandasecurity.com/packer-revolution`.

[5] CHEN, X., ANDERSEN, J., MAO, Z. M., BAILEY, M., AND NAZARIO, J. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *DSN'08: Proc. 38th Annual IEEE Int. Conf. on Dependable Systems and Networks* (June 2008), pp. 177–186.

[6] CHRISTODORESCU, M., KINDER, J., JHA, S., KATZENBEISSER, S., AND VEITH, H. Malware normalization. Tech. rep., Technische Universität München, 2005.

[7] COHEN, F. B. Operating system protection through program evolution. *Comput. Secur. 12*, 6 (1993), 565–584.

[8] COLLBERG, C., MARTIN, S., MYERS, J., AND NAGRA, J. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACSAC '12, pp. 319–328.

[9] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.

[10] COPPENS, B., SUTTER, B. D., AND MAEBE, J. Feedback-driven binary code diversification. *TACO 9*, 4 (2013), 24.

[11] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (2006), USENIX-SS'06.

[12] DYE, B. Symantec Develops New Attack on Cyberhacking. `http://online.wsj.com/news/articles/SB10001424052702303417104579542140235850578`, 2014.

[13] EL-KHALIL, R., AND KEROMYTIS, A. D. Hydan: Hiding information in program binaries. In *ICICS'04: Int. Conf. on Information and Communications Security* (2004).

[14] FERRIE, P. Attacks on virtual machine emulators. *`http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`* (2006).

[15] FLAKE, H. Structural comparison of executable objects. In *DIMVA'04*.

[16] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. H. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems* (1997), pp. 67–72.

[17] FRANZ, M. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proc. 2010 workshop on New security paradigms* (2010), NSPW '10, pp. 7–16.

[18] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not transparency: Vmm detection myths and realities. In *HOTOS'07: Proc. 11th USENIX workshop on Hot topics in operating systems* (2007), pp. 6:1–6:6.

[19] GEER, D., PFLEEGER, C. P., SCHNEIER, B., QUARTERMAN, J. S., METZGER, P., BACE, R., AND GUTMANN, P. CyberInsecurity: The cost of monopoly – how the dominance of Microsoft's products poses a risk to security. *Computer & Communications Industry Association Report* (2003).

[20] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 40–40.

[21] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. Ilr: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy* (2012), pp. 571–585.

[22] HOMESCU, A., NEISIUS, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization* (2013), CGO'13.

[23] HUANG, N.-F., AND TSAI, W.-Y. SHOCK: A worst-case ensured sub-linear time pattern matching algorithm for inline anti-virus scanning. In *ICC* (2010), IEEE, pp. 1–5.

[24] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential slicing: Identifying causal execution differences for security applications. In *IEEE S&P'11*.

[25] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (2007), WORM '07.

[26] KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. Emulating emulation-resistant malware. In *VMSec'09: Proc. 1st ACM Workshop on Virtual Machine Security* (2009), pp. 11–22.

[27] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *CCS'03: Proc. 10th Conf. on Computer and Communications Security* (2003), pp. 272–280.

[28] KISSERLI, N., CAPPAERT, J., AND PRENEEL, B. Software security through targeted diversification. In *Proceedings Third Int. Workshop on Code Based Software Security Assessments* (2007).

[29] KOJM, T. Clam antivirus (ClamAV). `http://clamav.net`.

[30] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *ACM Conference on Computer and Communications Security* (2011), pp. 285–296.

[31] KRÜGEL, C., ROBERTSON, W. K., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *USENIX Security Symposium* (2004), USENIX, pp. 255–270.

[32] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy* (2014).

[33] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04: Proc. 2004 Int. Symp. Code Generation and Optimization* (2004).

[34] LAU, B., AND SVAJCER, V. Measuring virtual machine detection in malware using dsd tracer. *Journal in Computer Virology* (2010), 181–195.

[35] LIN, Z., RILEY, R. D., AND XU, D. Polymorphing software by randomizing data structure layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2009), DIMVA '09, Springer-Verlag, pp. 107–126.

[36] LINDORFER, M., KOLBITSCH, C., AND MILANI COMPARETTI, P. Detecting Environment-Sensitive Malware. In *Recent Advances in Intrusion Detection (RAID) Symposium* (2011).

[37] 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).

[38] MOHAN, V., AND HAMLEN, K. W. Frankenstein: Stitching malware from benign binaries. In *WOOT* (2012), E. Bursztein and T. Dullien, Eds., USENIX Association, pp. 77–84.

[39] OBERHEIDE, J., BAILEY, M., AND JAHANIAN, F. PolyPack: an automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies* (2009), WOOT'09.

[40] O'KANE, P., SEZER, S., AND MCLAUGHLIN, K. Obfuscation: The Hidden Malware. *IEEE Security & Privacy 9*, 5 (2011), 41–47.

[41] OREANS TECHNOLOGIES. Themida advanced windows software protection system, 2013. `http://www.oreans.com/themida.php`.

[42] PALEARI, R., MARTIGNONI, L., FRESI, G., AND BRUSCHI, R. D. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *WOOT'09: Proc. USENIX Workshop on Offensive Technologies* (2009).

[43] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy* (2012), pp. 601–615.

[44] PAYER, M. Embracing the New Threat: Towards Automatically Self-Diversifying Malware. `https://nebelwelt.net/publications/14SYSCAN/`, 2014.

[45] PERDISCI, R., LANZI, A., AND LEE, W. Classification of packed executables for accurate computer virus detection. *Pattern Recogn. Lett. 29*, 14 (Oct. 2008).

[46] RAFFETSEDER, T., KRÜGEL, C., AND KIRDA, E. Detecting system emulators. In *ISC'07: Int. Conf. Information Security* (2007), pp. 1–18.

[47] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (2006), ACSAC '06.

[48] RUTKOWSKA, J. Red pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[49] SCHRITTWIESER, S., KATZENBEISSER, S., KIESEBERG, P., HUBER, M., LEITHNER, M., MULAZZANI, M., AND WEIPPL, E. Covert computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (2013), ASIA CCS '13, pp. 529–534.

[50] SOVAREL, A. N., EVANS, D., AND PAUL, N. Where's the FEEB? the effectiveness of instruction set randomization. In *SSYM'05: Proc. 14th Conf. on USENIX Security Symposium* (2005).

[51] STARFORCE TECHNOLOGIES. Aspack executable file compressor, 2013. `http://www.aspack.com`.

[52] SUTTER, B. D., ANCKAERT, B., GEIREGAT, J., CHANET, D., AND BOSSCHERE, K. D. Instruction set limitation in support of software diversity. In *ICISC* (2008), P. J. Lee and J. H. Cheon, Eds., vol. 5461 of *Lecture Notes in Computer Science*, Springer, pp. 152–165.

[53] SZÖR, P., AND FERRIE, P. Hunting for metamorphic. In *In Proceedings of the Virus Bulletin Conference, 2001 (VB '01)* (2001), pp. 123–144.

[54] TURKULAINEN, J. Reverse engineering malware binary obfuscation and protection, 2014. `https://noppa.aalto.fi/noppa/kurssi/t-110.6220/luennot/T-110_6220_binary_obfuscation_and_protection.pdf`.

[55] WILLIAMS, D., HU, W., DAVIDSON, J. W., HISER, J. D., KNIGHT, J. C., AND NGUYEN-TUONG, A. Security through diversity: Leveraging virtual machine technology. *IEEE Security and Privacy 7* (2009), 26–33.