# CSC 791 Evaluating Software Diversity Final Report (JHUAPL)

Aaron Fields, Dallas Wright
Dakota State University
{aaron.fields, dallas.wright}@dsu.edu

## Contents

## I. Project Summary

Software diversity has been suggested as a method of defense against 0-day exploits [1]. Many diversification methods has been proposed, but little is known about their effectiveness and/or side-effects. This project aims to compare the differences between Intermediary Representation (IR) and raw x86 disassembly language when used as feature vectors for clustering related binaries. The hope is that IR performs similarly (or even better) to raw x86, which could make developing a multi-platform clustering capability cheaper, if not free.

## II. Introduction

JHUAPL currently has the ability to cluster related binaries, but this capability is based on parsing raw x86 disassembly, and is therefore not supportive of other architectures, like ARM or MIPS. IR languages, such as VEX and LLVM, exist, but it is unknown what effect using an IR as a feature vector for a clustering algorithm will produce. Our hypothesis is that an IR interpretation of an executable binary will perform similarly, if not better than, the raw disassembly of an executable binary. This should allow us to quickly and easily develop a clustering capability for multiple platforms, since these IR languages can represent many different architectures by design.

### *A. Motivation*

As software reverse-engineering and exploitation becomes more sophisticated [2] and automated [3], software defense must take steps to remain a step ahead of software attackers. One such step that can be taken is the diversification of software to produce unique binaries for each target system. Unique binaries theoretically turn the attacker's task into a bit of a Catch-22 sort of problem: in order to exploit a target system and gain access, the target binary must be analyzed for vulnerabilities, but in order to get the correct binary to analyze, the attacker must exploit the system and obtain the unique binary. So, the motivation here is quite simple: we want to explore the effectiveness of software diversity as a software defense mechanism.

## III. Literature Review

Software diversity for defense is not a new idea [4]. Additionally, diversification techniques have been proposed (and implemented) in many different areas of operating systems and software. First, we should take a quick walk through software exploitation history. Basic memory corruption exploits take advantage of improper bounds checking, leading to buffer overflows on the stack, which allow an attacker to specify an arbitrary return address, leading to arbitrary code execution [5]. One response to this sort of attack was to introduce a randomized canary on the stack, which would be checked before returning [6]. If the canary had been changed (e.g. via a buffer overflow), the idea was that the program would exit safely. Implementations of stack canaries vary, but they are essentially an attempt to diversify software for defense.

Of course, attackers found that there were other control-flow constructs that could be overwritten and triggered, such as SEH pointers [7]. The response to this sort of attack was to introduce the *no-execute* (NX) bit, which could be used to mark sections of memory (the stack and the heap) as non-executable, which meant that attackers could no longer inject machine code directly. So, attackers moved to code-reuse attacks, commonly known as *return-oriented programming* (ROP) [8].

Because ROP attacks depends on being able to predict the location of useful memory locations, defenders sought to make them unpredictable via *address space layout randomization* (ASLR), which is a form of diversification implemented by the operating system at run-time [9]. ASLR randomizes the load address of modules in memory – a rather course-grained approach. Attackers are often able to circumvent ASLR by discovering information disclosure vulnerabilities that "leak" memory addresses, with just one leaked address giving away the memory location of an entire library. Attempts have been made to increase the granularity of ASLR [10] [11], but with arguably mixed results [12].

Stack canaries and ASLR are automated, run-time diversification techniques. Other techniques, such as *N-version programming*, are manual processes that require humans to use multiple programming languages, algorithms, designs, and implementations to produce multiple versions of a program that all meet a determined specification [13]. These sorts of techniques are known generally as *design diversification* and have historically been practiced as a method for increasing fault-tolerance in software during development [14], although they are not without their criticisms [15].

There have been studies that look at the effectiveness of software diversity for the purpose of fault-tolerance and reliability [16], but software diversity as a defense mechanism, specifically, is a more recent development [17]. These techniques are typically automated, and are usually performed at or near compile-time [17]. Automatic code *obfuscation* techniques are very similar to automatic software *diversification* techniques, but the goal of the techniques is different: obfuscation seeks to obscure the *input*, whereas diversification seeks to obscure the binary (e.g. machine code) *outputs* [18].

## IV. Methods and Procedures

With guidance from the sponsor, our focus has been narrowed into a more manageable scope since the initial proposal. Therefore, our plan/tasking differs a bit from what was originally proposed.

### A. Overview

*1) Obtain test corpus of related binaries:* We decided to use the GNU coreutils as our corpus.

*2) Obtain/implement a diversification technique:* We identified two open source, LLVM-based compilers that successfully produce diversified binaries: Multicompiler and obfuscator-LLVM. We only had time to work on output from Multicompiler.

*3) Automate feature vector generation:* This step includes automating the compilation, disassembly/IR generation, and any preprocessing necessary for feature vector generation.

During our research, we were able to develop four types of disassembly:

1. Raw instruction mnemonics with raw operands, very much like the output you would get from `objdump`.
2. Binned instruction mnemonics with raw operands.
3. Raw instruction mnemonics with binned operands.
4. Binned instruction mnemonics with binned operands.

Instruction mnemonic binning was based on the Capstone disassembler's instruction group metadata. This allowed us to bin instructions (i.e. all AVX instructions) into categories according to their type.

Similarly, operand binning was based on Capstone's operand metadata. Operands were binned into the following categories: `register`, `immediate`, `memory`, `invalid`, or `floating-point`.

*4) Experimentation & Analysis:* During this stage we used `scikit-learn` to experiment with different feature vectorizers, similarity metrics, dimensionality reduction algorithms, and clustering algorithms to produce a number of graphs for visual inspection.

For vectorization, the only method we have experimented with so far is TfidfVectorizer. This is often used in text analytics. It uses a bag-of-word approach to vectorize the features, and then normalized them based on Inverse Document Frequency (IDF). We measure similarity by calculating the cosine distance between vectors, then reduce the resulting cosine distance matrix to two dimensions via Multidimensional Scaling (MDS), and then plotted using MeanShift clustering.

### B. Deliverables

*All* code (including the code used for clustering and analysis) has been and/or will be organized, documented, and stored in a Git repository, which will be delivered to the sponsor.

### C. Limitations

*1) Limited sample set:* Our analyses used the GNU coreutils as our source for all datasets, the characteristics of which may have affected our results. The coreutils set is comprised of 103 programs, most of which are essentially single-purpose programs without graphical user interfaces.

*2) Time constraints/limited team size:* Between our small team size and the limited class duration, total man-hours were quite small, which severely limited the amount of testing and comparisons that we were able to do. For example, with more time, we could vectorize and cluster an IR representation of the binaries, examine the effects of using n-grams, and experiment with alternative similarity metrics and clustering algorithms.

*3) Not enough compute:* Compilation, disassembly, vectorization, and clustering are *all* intensive, slow operations that can derive a great speed benefit from multi-threading/multi-processing. We did not have access to any machines with more than 8 hyper-threaded cores, which meant many of our jobs took hours to run. For iterative processes like developing optimal feature vectors, any reduction to this time would have been helpful.

## V. Findings

Due to time constraints, we were only able to generate clusters using one pipeline, which consisted of the TfidfVectorizer, cosine distance metric, Multidimensional Scaling, and MeanShift clustering, in that order. Preliminary results appear to show that binning instruction mnemonics and/or operands produce tighter clusters of related binaries than raw mnemonics and operands. However, results are inconclusive, as we have not yet had time to analyze the clusters in depth.

## VI. Biographical Sketches

### *A. Aaron Fields*

- Software development in many programming languages, including "low-level" languages like C/C++.
- Reverse-engineering, utilizing various disassemblers, debuggers, and decompilers.

### *B. Dallas Wright*

- Software development in Assembler, C, C++ as well as many other higher level languages such as PASCAL, ADA, COBOL, etc.
- Project manager for Human Resource Systems, Department of Innovation and Technology, State of Illinois
- Supervise, design, development, implementation and maintenance of enterprise systems.
- Have written rudimentary compilers and linkers.

## Bibliography

[1] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 workshop on new security paradigms*, 2010, pp. 7–16.

[2] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE symposium on security and privacy*, 2016.

[3] J. Song and J. Alves-Foss, "The darpa cyber grand challenge: A competitor's perspective," *IEEE Security & Privacy*, vol. 13, no. 6, pp. 72–76, 2015.

[4] D. Partridge and W. Krzanowski, "Software diversity: Practical statistics for its measurement and exploitation," *Information and software technology*, vol. 39, no. 10, pp. 707–717, 1997.

[5] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *Usenix security*, 1998, vol. 98, pp. 63–78.

[7] D. Litchfield, "Defeating the stack based buffer overflow exploitation prevention mechanism of microsoft windows 2003 server," *Black Hat Asia*, 2003.

[8] H. Shacham, E. Buchanan, R. Roemer, and S. Savage, "Return-oriented programming: Exploits without code injection," *Black Hat USA Briefings (August 2008)*, 2008.

[9] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating systems, 1997., the sixth workshop on hot topics in*, 1997, pp. 67–72.

[10] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software." in *ACSAC*, 2006, vol. 6, pp. 339–348.

[11] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *23rd usenix security symposium (usenix security 14)*, 2014, pp. 433–447.

[12] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and privacy (sp), 2013 ieee symposium on*, 2013, pp. 574–588.

[13] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation,"

in *Digest of papers ftcs-8: Eighth annual international conference on fault tolerant computing*, 1978, pp. 3–9.

[14] B. Littlewood, P. Popov, and L. Strigini, “Modeling software design diversity: A review,” *ACM Computing Surveys (CSUR)*, vol. 33, no. 2, pp. 177–208, 2001.

[15] B. Baudry and M. Monperrus, “The multiple facets of software diversity: Recent developments in year 2000 and beyond,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[16] M. J. van der Meulen and M. A. Revilla, “The effectiveness of software diversity in a large population of programs,” *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 753–764, 2008.

[17] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated software diversity,” in *2014 ieee symposium on security and privacy*, 2014, pp. 276–291.

[18] R. Pucella and F. B. Schneider, “Independence from obfuscation: A semantic framework for diversity,” in *19th ieee computer security foundations workshop (csfw’06)*, 2006, pp. 12–pp.