Frankencode: Creating Diverse Programs Using Code Clones

Hayley Borck, Mark Boddy, Ian J De Silva, Steven Harp,
Ken Hoyme, Steven Johnston, August Schwerdfeger, and Mary Southern
Adventium Labs, Minneapolis, MN 55401
Email: firstname.lastname@adventiumlabs.com

Abstract

In this paper, we present an approach to detecting novel cyber attacks though a form of program diversification, similar to the use of n-version programming for fault tolerant systems. Building on extensive previous and ongoing work by others on the use of code clones in a wide variety of areas, our Functionally Equivalent Variants using Information Synchronization (FEVIS) system automatically generates program variants to be run in parallel, seeking to detect attacks through divergence in behavior. Unlike approaches to diversification that only change program memory layout and behavior, FEVIS can detect attacks exploiting vulnerabilities in execution timing, string processing, and other logic errors.

We are in the early stages of research and development for this approach, but have made sufficient progress to provide a proof of concept and some lessons learned. In this paper we describe FEVIS and its application to diversifying an open-source webserver, with results on several different example classes of attack which FEVIS will detect.

1. Introduction

Software vulnerabilities and the cyber attacks enabled by them are increasingly prevalent, and continue to increase in both cost and impact. Critical civilian and military software systems are vulnerable, with new vulnerabilities being discovered all the time. Vulnerabilities enabling attacks can remain undetected for years, as with the Heartbleed bug in OpenSSL [1]. Worse, both known and unknown vulnerabilities may enable not just new attacks, but new kinds of attacks. Apparently-minor vulnerabilities can be chained together in successful attacks, even on software that is extensively tested and generally viewed as secure.[1]

In this paper, we describe our research on the Functionally Equivalent Variants using Information Synchronization (FEVIS) system. Part of the Cyber Fault-tolerant Attack Recovery (CFAR) program, funded by the U.S. Government's Defense Advanced Research Projects Agency (DARPA), FEVIS builds on previous and ongoing work on "code clones," substituting redundant code fragments as a means to generate program variants automatically. These variants are intended for use in a multi-variant execution environment, to be used for attack detection and resistance in the presence of both known and unknown attacks. We are in the early stages of research and development for this approach, but have made sufficient progress to provide a proof of concept and some lessons learned. In the rest of this paper, we describe related work and previous research on which FEVIS builds (Section 2). We then present FEVIS itself (Section 3), illustrating the approach through the use of examples drawn from the diversification of an open-source webserver (Section 4). We conclude with a discussion of ongoing and future research.

2. Related Work

FEVIS applies concepts and techniques from the study of code redundancy as a means of diversifying software for use in multi-variant execution. In this section, we summarize related work in these three areas.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-15-C-0110. The views and/or findings contained in this article are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the US Government. DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

1. The multi-step attacks on Chromium detailed in "A Tale of Two Pwnies" provide good examples [2].

2.1. Multi-Variant Execution

Multi-variant execution is an approach to detecting and forestalling attacks, in which multiple variants of an application are run in parallel. These variants are specifically chosen to provide the same results for an authorized range of inputs, but to diverge under attack. Multi-variant execution has been an active research topic for several years, with different research groups proposing alternative architectures for variant synchronization and checking, as well as methods for variant generation including Address Space Layout Randomization (ASLR); Intelligence, Surveillance, and Reconnaissance (ISR); and buffer padding. For example, Salamat *et al.* [3] describe a Multi-Variant Execution Environment (MVEE), using variants in which the stack grows in opposite directions. Nguyen-Tuong *et al.* devised a multi-variant system using data transformations derived from N-variant systems [4].

Multi-variant execution is somewhat different from the use of diversification as a "moving target defense" (MTD), in which the objective is to present an attacker with a single binary, but one which has unpredictable differences from previous instances of the same program they may have investigated. One critical difference is that multi-variant execution is *secretless*, in the sense that an attacker may know about how variants are generated, may even know which specific variants are currently running, without that information enabling an attack. In contrast, an attacker given time to investigate a single variant, may discover what has been changed and be able to adjust their attack accordingly.[2]

2.2. Program Diversification

One common thread through much previous research on program diversification for multi-variant execution is the preservation of program semantics in a very strong sense. ASLR relocates code and data in the running program, preserving control flow. The same is true for techniques such as stack or heap padding, or ISR. Program transformation approaches that seek to defeat attacks upon a single binary, such as the addition of stack canaries, might plausibly be viewed as providing diversification. In this case, the altered and unaltered programs constitute a set of two.

However, many of the most common exploits, enabling the most devastating attacks, exploit holes in the logical structure (i.e., the semantics) of the program itself. Cross-site scripting, use after free, directory traversal, SQL and OS command injection, and unauthorized disclosure via URL string processing vulnerabilities are among the many forms of attack that may not be blocked or detected by diversification measures that seek to preserve program control flow and semantics. Nor will many of them be caught by more sophisticated approaches such as automated code obfuscation or restructuring, if the semantics of the modified program remain the same.

These are the kinds of attacks that we seek to detect using variants generated by FEVIS.

2.3. Code Clones

Our approach takes as a starting point the well-known concept of *code clones*, in which redundant code sections can be substituted one for another. The definition of redundancy that we follow is taken from Carzaniga *et al.* [6], who define code redundancy as the product of *observational equivalence* and *execution distance*.[3] Observational equivalence is defined in terms of *probing code*, code which is input to code fragment to determine outputs and an *oracle*. Both are appended to the code fragments being tested for equivalence. This definition allows us to define "functional equivalence" in terms of authorized or expected input, rather than across the entire space of possible input. Carzaniga *et al.* evaluate several definitions for execution distance, settling on *data projection*, which is measured in terms of the difference in memory locations read and written, as well as what is written to them. Figure 1 provides two examples of redundant code fragments according to this definition.

Code clones have been employed for a wide variety of uses, including the automatic generation of test oracles, self-healing code, fault tolerance, and automatic test generation. For example Carzaniga [8] creates test oracles by finding redundancies in code and cross checking the execution of a test with the execution of the same test on a variant of the code with redundant operations replaced.

Clones may either be discovered or constructed, both approaches having some currency in the research literature. The prevalence of redundant code fragments in large software systems such as the Linux kernel has been extensively documented, as discussed in [8],

2. Larsen *et al.* [5] provide a good summary of diversification methods applied as MTD.

3. This is distinct from the more general use of the term "clone" as used (for example) in [7], where clones may differ not at all, or only in layout and choice of identifiers. Clones in this larger sense have been applied for purposes such as detecting plagiarism and copyright infringement.

linkage: int x; int y;	

int tmp = x;
x = y;
y = tmp;
	x ^= y;
	y ^= x;
	x ^= y;

linkage: AbstractMultimap map; String key, Object value;	

map.put(key, value);	List list = new ArrayList();
	list.add(value);
	map.putAll(key, list);

Figure 1. Redundant code fragments, from [6]

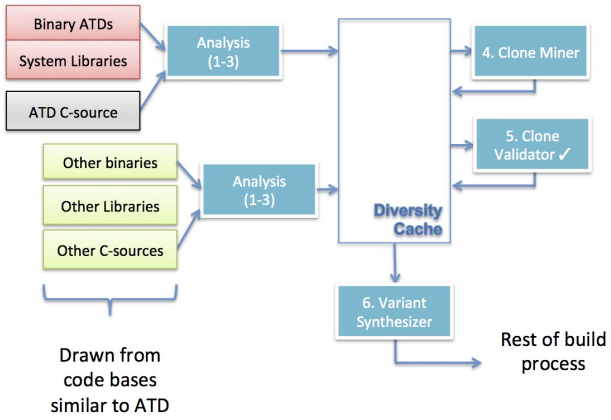

Figure 2. FEVIS Architecture Overview

[9]. Automatically finding these redundant fragments is an ongoing topic of research. The EQMiner tool of Jiang and Su uses random input and output testing on arbitrarily sized code snippets to find functionally equivalent yet syntactically different code clones [9]. The semantic clone detection tool MeCC [10] compares abstract memory states in order to find candidate code clones.

Goffi *et al.* [11] use genetic algorithms to generate functionally equivalent clones. This approach sidesteps the difficulties of finding equivalent code fragments in the existing code base, but at the cost of potentially limiting the space of redundancies explored.

3. FEVIS

The approach supported by FEVIS is to take an *Application to Defend (ATD)*, identify a set of *ATD segments* for possible replacement by functionally equivalent *variant segments*, resulting in a set of *ATD variants*.

A high-level view of the FEVIS system architecture is shown in Figure 2. For a given ATD, the process supported by this architecture is as follows:

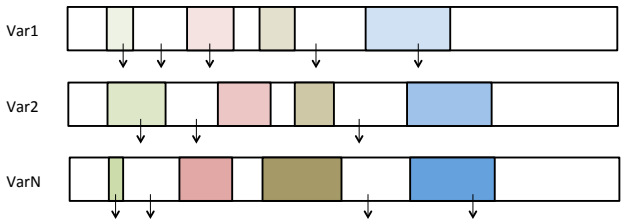

Figure 3. ATD variants shown running in parallel

1) Partition the ATD into chunks (potential variant segments)
2) Identify potential clones for those chunks:
   a) Code chunks previously encountered and stored in a "Diversity Cache"
   b) Redundancy within the ATD itself
   c) Chunks from related implementations in other applications with similar functionality
   d) Synthesized variants
3) Find or synthesize adaptors for chunks identified in Step 2
4) Generate ATD variants by swapping in clones for one or more variant segments
5) Generate arguments supporting ATD variant functional equivalence and divergence

The current implementation encompasses automated "chunking" of the ATD and the manual identification of potential variant segments within the Diversity Cache, along with the appropriate adaptors, and then automatic generation of a set of ATD variants. Tools for automated detection of potential code clones are under construction, as is the automated generation of arguments regarding the properties of a given set of ATD variants.

Figure 3 shows several variants for the same ATD. The shaded regions represent variant segments. The small vertical arrows represent system calls that may be made within those segments, and are included to show that rigorous synchronization of the variants at system calls will generate numerous false positives.

We use the term *adaptor* to refer to code that may be wrapped around a given code fragment, in order to make it equivalent to an ATD segment. For example an adapter may be needed to swap the arguments for code fragments with equivalent arguments in different orders. Figure 4 shows how an adaptor may be combined with a code chunk from the Diversity Cache, in order to generate a variant segment that is functionally equivalent to some ATD segment. While in the general case finding adaptors is equivalent to program generation, in limited form it appears to show promise, especially for segments defined at function

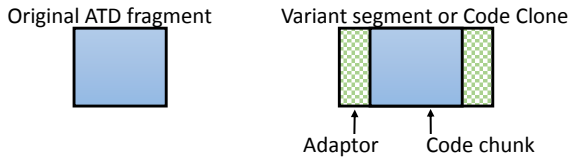

Figure 4. Diagram of a clone: an adaptor wrapper around a code chunk

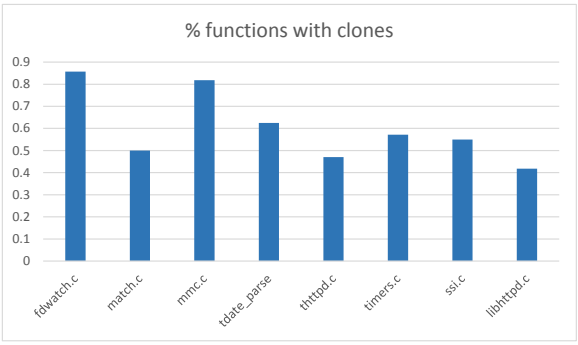

Figure 5. Percent of the functions within each file that comprises the thttpd webserver for which clones were found for in a manual search.

boundaries.

4. Progress to Date

At this early stage of the project (within the first year), we have implemented the basic functions for FEVIS. For motivation and experimentation, we have investigated the open-source webserver thttpd, searching for code clones and evaluating the efficacy of ATD variants for diversification.

In a manual search for variant segments for thttpd, we found that 51% (Figure 5) of the code base had potential clones in other open source webservers (Apache, Hiawatha, Cherokee, and lighttpd were examined). Among the results this analysis turned up were potential clones for the functions defang, strdecode, match_one, tdate_parse, atoll, and fdwatch, with the number of alternative choices ranging from 1 to 5.

In some source files within thttpd, as many as 80% of the functions had clones elsewhere, or even within the same file. In addition, we explored the use of alternative library implementations (for example, the many available versions of the C standard library) as a further source of diversity.

In tests conducted within the overarching DARPA program, sets of ATD variants automatically generated by FEVIS were shown to diverge under multiple kinds of memory-based attacks, using test sets provided by a third party. Sets of two to three variants were created using the manually found clones. Attacks such as heap and stack based buffer overflow and out of bounds read were tested, which are each in the top ten CWE vulnerabilities list. We have additionally demonstrated the generation of FEVIS ATD variants that will diverge under attacks exploiting program semantics. For this experiment, as the ATD we used an older version of the lighttpd webserver with a documented security flaw (CVE-2005-0543). The vulnerability involves the processing of submitted URLs: due to the improper handling of escaped null characters (%00) an attacker can obtain the source code for a CGI script, rather than just the results of a query submitted to that script. As these scripts frequently contain hard-coded URLs, directory paths, and even login credentials, this is a significant vulnerability.

Potential clones for this vulnerable variant segment were found in four alternate webservers: thttpd, Hiawatha, Apache, and Cherokee. The potential clones from thttpd and Hiawatha had the same vulnerability; those from Cherokee and Apache did not, though they fixed it in different ways; Cherokee mapping null characters to spaces, and Apache returning an error.

However, while the analogous string-handling function in thttpd mishandles nulls in the same way as lighttpd, the thttpd webserver was not exploitable in the same way, due to the details of subsequent processing. This, along with the differences in string processing in the Cherokee and Hiawatha webservers described above, provides some preliminary evidence for our claim that it is not necessary for a given vulnerability to appear in a variant segment, in order to have a set of ATD variants diverge when an exploit is attempted.

5. Conclusions and Future Work

The question motivating this research is whether creating a diverse set of variants of a program from functionally equivalent code clones will enable broader detection of cyber attacks. We are still within the first year of this project, and so our results are necessarily preliminary, but the initial indications are promising. In this section, we discuss ongoing and future work on FEVIS.

The primary areas for improvement are in automatically finding or generating clones, improved automated testing for equivalence among variant segments (including the generation of adaptors where needed), and the automated construction of arguments regarding the functional equivalence and divergence properties of sets of ATD variants.

In addition, we seek to be able to work from binaries, as well as from source code. Currently we use source compiled to LLVM [12] within the Diversity Cache. Working from binaries lifted to LLVM is more difficult because lifted binaries have had much of the structure and typing information removed in the compilation process. One possible workaround for this problem is to work directly from disassembled binaries, which somewhat paradoxically have more structure than the lifted representations, along with the use of tools to recover some of the structure lost in the compilation process.

For automated search for potential clones, we are working two approaches. The first is to adapt the approach implemented in EQMiner to work on LLVM. The second is to further exploit the very considerable redundancy present in standard libraries. Presently, we can and do swap entire libc versions into ATD variants. Piecemeal substitution of individual library functions will provide much greater freedom to generate variants, but comes with some additional complications, such as identifying functions that must be swapped together, because they make common use of shared information or data structures. Automated testing for functional equivalence is also required. Previous work on clone discovery such as EQMiner uses randomly-generated test strings [9]. More directed testing can be accomplished using a cached set of probes generated from "normal operation" of a code fragment (*e.g.*, gathered using Daikon [13]), augmented over time with counter-examples from more detailed subsequent testing.

In collaboration with our partners on this project at the University of Minnesota, we are also investigating a hybrid approach in which adaptors are automatically synthesized (within a significantly limited search space) in order to make a given code chunk equivalent to some ATD segment. This work is promising, but too preliminary to present as yet.

Finally, given the extremely large space of ATD variants that could be generated from substituting for even a modest number of ATD segments, we must be able to construct much smaller sets of ATD variants that are likely to diverge under attack (necessary for attack detection), while continuing to exhibit equivalent behavior under normal conditions (necessary for user acceptance, and for avoiding false positives). Also in collaboration with the University of Minnesota, we are exploring ways to the construction of such arguments.

References

[1] "CVE-2014-0160." Available from MITRE, CVE-ID CVE-2014-0160., Dec. 3 2013. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[2] J. L. Obes and J. Schuh. (2012) A tale of two pwnies (part 1). [Online]. Available: http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html

[3] B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime defense against code injection attacks using replicated execution," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 4, pp. 588–601, 2011.

[4] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, "Security through redundant data diversity," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 187–196.

[5] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *Security & Privacy, IEEE*, vol. 12, no. 2, pp. 28–35, 2014.

[6] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proc. of Int. Conf. on Software Engineering (ICSE)*, 2015.

[7] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Technical Report 541, Queens University at Kingston, Tech. Rep., 2007.

[8] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 931–942.

[9] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 81–92.

[10] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 301–310.

[11] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella, "Search-based synthesis of equivalent method sequences," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 366–376.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[13] M. D. Ernst, "Dynamically discovering likely program invariants," Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.