FrankenSSL: Recombining Cryptographic Libraries for Software Diversity

(Invited Paper)

Bheesham Persaud, Borke Obada-Obieh, Nilofar Mansourzadeh, Ashley Moni, Anil Somayaji
School of Computer Science, Carleton University
Ottawa, Ontario, Canada
{bheesham, borke, nilofar, ashley, soma}@ccsl.carleton.ca

***Abstract*—Many security vulnerabilities arise from protocol implementation flaws. Software diversity can reduce the impact of such flaws; however, in practice there are relatively few implementations of important protocols due to the challenge of making interoperable, reliable, and efficient implementations. One strategy for increasing the number of variants is to mix and match components from different implementations. Just as biological systems recombine DNA from parent organisms to create children, we could mix and match code from different protocol implementations to get thousands of variants. To achieve this goal, two fundamental challenges must be overcome: we need to demonstrate that these variants can be created while preserving functionality, and we need to show that these variants are not all susceptible to the same vulnerabilities. As a step towards this goal we are developing a method for recombining implementations of Transport Layer Security (TLS), specifically OpenSSL, LibreSSL, and BoringSSL. In this paper we report on our progress to date.**

***Keywords*—software diversity, software recombination, library security**

I. Introduction

Each year, large numbers of computer security vulnerabilities are reported as well as strategies to exploit them. To address such discoveries, most Internet-connected platforms have extensive resources devoted to developing and deploying patches for software vulnerabilities. Despite such efforts, however, many systems remain unpatched, and even patched systems are subject to exploits of zero-day vulnerabilities. While formal methods hold the promise of provably secure systems, to date there has been little success in developing Internet-connected systems that have no exploitable vulnerabilities.

One approach is to reduce the predictability in how systems behave so as to make it harder to exploit vulnerabilities. Runtime randomization defenses such as ASLR [1] make it harder to perform memory corruption-based (buffer overflow) attacks. Strategies such as derandomization [2] and heap spraying [3], however, can be used to circumvent runtime randomization. With runtime randomization, the attacker can largely determine the likelihood of success or failure on a targeted machine simply by studying the behavior of a test machine under their control running the same software. If software were truly diverse, however, attackers would lose this advantage because the software running on any test machine would (with high probability) be different from the software running on the target.

The dangers of "software monocultures," or the lack of software diversity, have been noted by many researchers [4], [5]. Almost all work in software diversity, however, has been focused on the automatic creation of software variants at levels not specified in program source code [6], [7]. This style of diversity can be effective against attacks that make use of regularities in program memory layout and behavior (such as buffer overflows). Yet there exists many vulnerabilities which do not make use of such regularities but are very exploitable. Whether it be incorrect authentication checks, race conditions, protocol implementation errors, or even more idiosyncratic exploitable mistakes, such vulnerabilities cannot be mitigated though semantic-preserving changes in how code is compiled or run because the problems are in the programs' semantics.

A diversity-style approach to mitigating such vulnerabilities thus must make use of different implementations of the same functionality. Creating independent implementations of critical functionality is a practice long followed in the fault-tolerance community [8], [9]. Today there exist multiple open-source implementations of critical Internet infrastructure, server applications, and even many desktop applications such as web browsers. While the presence of these variants provides a small amount of diversity, an attacker still gets a lot of benefit from exploiting popular variants because they are installed on millions of systems.

Thus to improve security through diverse implementations we need more than a handful of variants; instead, we need enough variants that it is not feasible for attackers to study and develop exploits for each one. We cannot hope to create so many variants manually, and we cannot create them by simply randomizing the behavior of individual implementations (as has been done in previous work). The key insight of this work is that we could combine portions of implementation with one another to create diversity. Done properly, we would get a combinatorial explosion of variants that would be challenging for an attacker to characterize, let alone exploit.

Differences in program structure and function would suggest that this would be a fool's errand, as any arbitrary mixing of code from different sources would most likely result in a non-functional program. The entire field of genetic programming, however, is based on creating new programs by recombining fragments of other programs [10]. Further, in past work in our group we have had success with the automatic recombination

of regular program binaries at the object-file level [11], [12]. To get significant improvements in diversity that have measurable improvements in security, however, we need to maximize implementation diversity while also preserving higher-level semantics, particularly at the network and user interface level, and do so with existing programs—something past approaches cannot do.

Rather than recombining portions of entire programs, here we propose to recombine portions of security-critical libraries. Libraries present a stable API to applications yet can differ in their internal semantics. So long as libraries implement the same API, they may be interchanged. Thus, so long as we can create library variants that implement the same API we can get most of the benefits of implementation diversity at much lower cost.

As a first test case, we are working on recombining libraries implementing standard SSL/TLS functionality, as such libraries are both critical for security-related functionality and have been shown to have many exploitable vulnerabilities. Here we report on our progress on recombining elements of OpenSSL [13] with two OpenSSL forks, BoringSSL [14] and LibreSSL [15], [16]. As we will explain, recombination can be challenging even between libraries that have recently diverged from each other. Nevertheless, results to date indicate that our approach has promise.

The rest of this paper proceeds as follows. We first describe background and related work in Section II. We discuss our approach in more detail in Section III. Sections IV & V present details of our design and implementation progress, respectively, while Section VI reports on our preliminary results. Section VII discusses limitations and plans future work; Section VIII concludes.

II. Background

A. Software Diversity for Security

The idea of software diversity was proposed in fault tolerant systems as far back as the late 1970s, in the form of N-version programming [8]. In 1993, Cohen et al. [4] also recognised the importance of software diversity for software security and reported potential dangers of carrying out "software monocultures."

While using teams of developers to create independent implementations has been proposed in the context of security [17], the prohibitive costs of doing so have generally precluded this approach. Instead, most work on software diversity for security has followed the direction described in 1997 by Forrest et al. [6] where compile time and runtime diversity (really, randomization) are added in ways that preserve the source code level semantics of the application [7]. Most notably, address-space layout randomization (ASLR) [1] is now a common feature of modern operating systems; however, randomization approaches have also been proposed at virtually all levels of program behavior including the instruction set [18], [19], memory address [20], function layout [21], and system calls [22], [23]. Randomization-based defenses are often criticized as they only provide probabilistic protection; however, through N-variant programming [24] where the execution of variants are compared at runtime, it is even possible to get deterministic guarantees.

Because these transformations preserve source code-level semantics, they are primarily useful in detecting memory corruption attacks. Randomization at higher levels is also possible, however, in order to mitigate attacks such as SQL attacks [25] and cross-site scripting attacks [26]. But again, such defenses cannot mitigate vulnerabilities that arise from more general classes of implementation mistakes or design errors.

B. Software Recombination & Patching

While most code is created manually by programmers, there is a large body of work on automated ways to create code. Compilers and assemblers are of course automated code translators and generators; further, any declarative programming system directly or indirectly generates code. By their nature, however, these types of systems produce systems with relatively little diversity.

Natural evolution has proven to be very good at automatically producing diverse systems, so in considering ways to automatically generate diversity it makes sense to look to systems for evolving code. Most work in genetic programming (GP)—the main community concerned with evolving code—has focused on the evolution (really, searching in a pre-defined space) of code which satisfies various fitness functions [10]. While there is work on diversity in genetic programming, such work is mostly concerned with improving the quality of GP search rather than creating diverse solutions for their own sake [27], [28], [29]. While most work on genetic programming only works with S-expressions, not the source or object code of standard programming environments, there has been progress in using GP to automatically create patches to repair security vulnerabilities [30].

The work closest to ours here is that of Foster and Somayaji's ObjRecombGA (2010) [11], [12]. ObjRecombGA used a genetic algorithm to search for ways to successfully recombine object files between two closely related programs to create (with the help of a specialized linker) a new program that combines the functionality of the two "parent" programs. This approach was successful in combining functionality between variants of open source programs including a UNIX command line program (GNU sed), a web browser (Dillo), and a game (Quake). While object-level recombination may hold promise for creating variants for security purposes, ObjRecombGA tends to create variants that are mostly one version of the program with small additions from the other parent. It does this in order to minimize the linking problems caused by symbol, data structure, and function declaration mismatches between object file variants.

Thus, to create a larger number of variants, we need to study and develop techniques for managing the incompatibilities that arise between divergent code bases.

C. OpenSSL Security

The OpenSSL library started in 1998 as an implementation of TLS. Over time, however, it has become a comprehensive library implementing a wide range of cryptographic primitives and protocols that a developer might require. While OpenSSL is widely used, its security record has been problematic at best. In 2014 and 2015 the OpenSSL project reported a total of 54 Common Vulnerabilities and Exposures (CVE). As of May 2016, many more vulnerabilities have been documented in OpenSSL [31]. While many of the past vulnerabilities in the OpenSSL library have been things such as buffer overflows, integer overflows, and other types of memory corruption attacks, many others were implementation errors that are not easily mitigated through an implementation-level randomization approach. Consider these examples:

- CVE-2003-0147 (OpenSSL Advisory) 14th March 2003: RSA blinding was not enabled by default, potentially allowing attackers to obtain the server's private key through a timing attack.
- CVE-2008-0166 (Debian Advisory) 9th January 2008: OpenSSL on Debian systems generated predictable random numbers due to a change made to remove warnings generated by Purify and valgrind [32].
- CVE-2008-5077 (OpenSSL Advisory) 7th January 2009: Incorrect checking of a return result caused bad signatures to be treated as being correct.
- CVE-2014-0160 (OpenSSL Advisory) 7th April 2014: A critical buffer over-read bug was discovered and named Heartbleed [33].

The above vulnerabilities are all attributable to idiosyncratic implementation mistakes. Different implementations would be unlikely to have made exactly the same mistakes.

D. OpenSSL Forks

Soon after the Heartbleed bug in OpenSSL [33] was disclosed, OpenSSL was forked by two groups looking to improve the code quality of the project: LibreSSL, managed by OpenBSD developers [16], and BoringSSL, managed by Google developers [14].

LibreSSL has been designed as a streamlined drop-in replacement for OpenSSL, implementing the same basic API and much of the same functionality, although many uncommonly used and insecure algorithms have been removed. In contrast, BoringSSL, while also streamlined, has not been intended as a drop-in replacement; instead, the BoringSSL developers have been willing to change APIs and functionality to improve security, performance, and usability. The BoringSSL developers at Google are willing to do such changes because they also control the codebases that use BoringSSL. Nevertheless, the basic APIs provided by BoringSSL are still largely identical to those provided by OpenSSL.

Often when forks are created the original project loses resources; however, around the same time as BoringSSL and LibreSSL were founded, the OpenSSL project also received an influx of developer attention and funding. Thus, OpenSSL has proven to be a kind of "natural experiment" in the creation of security-critical code diversity with multiple, mostly compatible implementations now in existence.

III. Library-level Software Recombination

In order to create software diversity that will improve software security, here we propose to focus our efforts on creating diverse implementations of security-critical libraries rather than diversifying entire applications. Our strategy follows that of ObjRecombGA [11] in that we wish to recombine the code from different implementations. In order to get finer-grained recombination (and hence more variants) our aim is to enable recombination at the function level rather than the object file level.

We propose to focus on library-level recombination for multiple reasons.

- Library-level vulnerabilities can affect many applications; thus, efforts to improve the security of such libraries will tend to have disproportionate benefit.
- Important libraries often have multiple implementations, and these implementations are often closely related. (For example, libraries are often forked in order to satisfy the requirements of large applications.) Even when these implementations do not provide identical APIs, they are still often substitutable with modest effort.
- Fine-grained recombination will likely require some amount of manual effort. By focusing on libraries we can leverage this effort to improve the security of multiple applications.

FrankenSSL thus is a first effort to do library-level software recombination between OpenSSL library and recently created forks of OpenSSL. As explained in the last section, OpenSSL is widely used, security-critical, has a long history of vulnerabilities, and there exist high quality forks of OpenSSL.

IV. Design

While the basic principle behind FrankenSSL's design can be translated to any other library with multiple forks or implementations, for clarity here we describe our design in terms of three libraries, OpenSSL, BoringSSL and LibreSSL.

A library can be modeled as an abstract cluster of functions that all interlink. This abstraction is not that far from the reality of their representations in a filesystem as sets of object files with symbolic references to each other.

Applications that wish to use SSL/TLS normally interact with their library of choice through APIs. The functions that represent those libraries also interconnect with each other and use each other through internal APIs, since modern software engineering prioritizes modular over monolithic architecture for functionality.

Normally a target application would link (through its APIs) to a set of functions within the function cluster of the library, and those functions would link to other functions or each other. This pattern creates a straightforward hierarchy of abstraction, from the application (that asks for data to be encrypted or

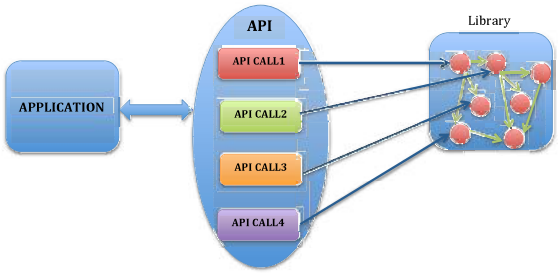

Fig. 1. Applications normally link into libraries through an established graph of dependencies.

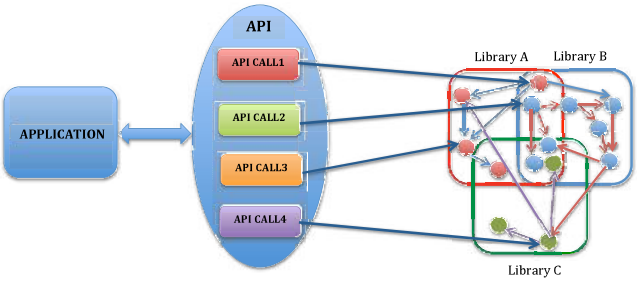

Fig. 2. Our goal is to layer different implementations of the same library to allow wider functional diversity through procedural cross-linking.

decrypted) down to terminal functions in the graph (that manipulates and manages data and memory).

The three implementations of SSL can be considered to be three clusters of functions, with some overlap. Since they are related to each other (with BoringSSL and LibreSSL both being forks of OpenSSL), some functionality is shared across multiple libraries. Each effective library can be modeled mostly as a directed graph, starting from the application and splaying out into the shared function cluster. See Figure 1.

From this formalization, it's easy to see that alternate potential implementations of SSL can simply be modeled as unique graphs. FrankenSSL's goal is to procedurally generate random, viable API/linker graphs upon this shared library of functions and symbols. See Figure 2.

Exploits in any of the implementations are ultimately located in a specific function, or in the interplay between a small set of functions. As a consequence, randomized implementations drastically cut down on the effective portability of any given exploit, since an attacker can't guarantee that any given system has the relevant function or function constellation. Even if a given implementation does have the target functions in the target arrangement, alterations to the rest of the graph may lead to the runtime being structured differently in a manner reminiscent of Address Space Layout Randomizaton [1].

V. Implementation Progress

While we eventually would like to recombine libraries at arbitrary function boundaries, initially we have focused on recombining the libraries at the level of calls that are commonly made by applications into the library. Applications created using OpenSSL (and thus BoringSSL and LibreSSL) generally follow the same workflow: initialize the library, initiate a TLS session, configure the TLS session, read from a session, write to a session, deinitialize the session, and then proceed to deinitialize the library. We wish to distribute these calls across the different libraries: we want to initialize with OpenSSL while reading from a session using LibreSSL's functionality, for example, as well as distributing internal calls across implementations.

As a first step towards this goal, we have focused on recombining the initializaiton and deinitialization code of each of the libraries. Specifically, we focused on recombining calls to the following functions:

- `SSL_load_error_strings()`,
- `SSL_library_init()`,
- `TLS_client_method()`,
- `SSL_CTX_new()`,
- `SSL_CTX_free()`, and
- `SSLeay_version()`.

In the rest of this section we describe the project scaffolding, build system, and linking issues. In the next section we present the results of these efforts.

A. Scaffolding

Installing OpenSSL, BoringSSL, and LibreSSL side-by-side on the same system can be a daunting task for novices. Each of the libraries—due to the nature of having the same origin—export similar symbols, build artifacts with the same name, and install to the same directories.

Installation of any one implementation on a system-wide basis would mean that another implementation cannot be used due to the resulting collisions. This makes having an installation of more than one implementation based on OpenSSL difficult, to say the least. These collisions make it impossible to link all three into the same application without any edits to any of the implementations.

Our solution is to link applications against a hand-crafted dynamically linked stub library. We then specify different combinations of implementations at runtime using `LD_PRELOAD`. This method gets rid of any high-level namespace collisions, however, it only works if all three libraries can be linked in to the application at once. To allow this to happen we had to change how the libraries were built.

B. Build Infrastructure

One of the changes that both BoringSSL and LibreSSL made from OpenSSL was a change in build infrastructure. OpenSSL makes use of a custom build infrastructure based upon standard Makefiles. BoringSSL and LibreSSL, however, were changed to use CMake.

Initially we tried converting the build system for OpenSSL to CMake. This proved to be challenging. OpenSSL requires many files to be generated before compilation and a number of compiler flags to be passed in at configuration time. Many of these flags are not documented, and as such are difficult to account for when migrating the build system.

Rather than migrate all of the code to a single build system, we instead modified each as needed in order to allow code from all three libraries to be incorporated into a single binary. To accomplish this we had to address the symbol resolution problem.

C. Symbol Resolution

Versioning the shared objects allows maximal modularity as we can combine exported symbols from each linked library at runtime. While OpenSSL versions symbols by default, BoringSSL and LibreSSL do not; thus, BoringSSL and LibreSSL export the same symbols for various functions, preventing them from both being linked in to the same binary.

Before we could implement a wrapper around each library we needed to ensure that the correct function would have been called from each library. That is, we needed to make sure that the program, at runtime, knew which version of a function to call.

We investigated multiple solutions to get around symbol collisions, namely: the use of objcopy to rename important symbols, writing a compiler pass, and using symbol versioning.

- **objcopy:** Using this method would require us to modify not only the compiled shared object files, but also to create our own header files with prototypes to each of the functions. This would, in essence, give a result equivalent to writing our own compiler pass.
- **Compiler pass:** This method included creating a plugin for GCC or writing an LLVM pass to rename symbols of each library at compilation time such that there would be no colliding symbols. However, writing a plugin for GCC or a pass for LLVM would require anyone who would want to build the software to have their compiler configured correctly. While not unreasonable, it is also not ideal as it would introduce complexities that come with working at the compiler level.
- **Symbol versioning:** After reading the section on Symbol Relocations, and Export Control of Drepper's "How To Write Shared Libraries" [34], we decided that his method was perfect for our needs. It required an additional two compiler flags and the creation of a symbol map which was trivial to write. By exporting the version of a symbol, each library we linked to would have saved a reference to the library it was referring to at link-time, which meant that at runtime the expected functions should be called.

Ultimately, we moved forward with symbol versioning as it is, we believe, the most transparent way to achieve non-colliding, globally exported symbols and only requires minimal changes to each codebase.

D. Runtime Linking

Each method that is wrapped from OpenSSL, BoringSSL, and LibreSSL is packaged into its own shared object file. While this generates three times as many files as wrapped methods, it safely separates each method from each library.

We link each method from the desired library to a test program which uses each of the methods. Linking is done at runtime by specifying each wrapper through the use of the `LD_PRELOAD` environment variable. If the program runs without any errors, it is considered to have a successful combination of methods.

VI. Preliminary Results

We have tested all combinations for intialization and deinitialization using the three libraries and none of them are viable for practical use. Programs have either thrown a fatal error or have leaked memory—not ideal in either case. The memory errors arise because of the difference in the `SSL_CTX` structure defined in each library. This causes complications as the `SSL_CTX` structure is used to store state throughout each session.

We created a simple application to test whether or not a combination was viable. Any combination ran by the program had to go through the initialization and deinitialization processes without errors to be considered a viable combination. Errors during runtime were checked for using valgrind and GDB.

TABLE I
Results from manual testing

init	deinit	Result
Open	Open	Works
Open	Boring	SIGABRT
Open	Libre	SIGSEGV
Boring	Boring	Works
Boring	Libre	Leak
Boring	Open	Leak
Libre	Libre	Works
Libre	Boring	SIGABRT
Libre	Open	SIGSEGV

Read and write operations in the sessions were not supported by any combination that was not error-free; thus, we omitted these results from the above table.

There was no case in which any of the libraries worked well together, even with some rudimentary "shim" code. The differences between each library's SSL context, and what is required to correctly allocate it vary too greatly to work without significant amounts of interface code.

VII. Discussion

We have attempted to recombine parts of OpenSSL, LibreSSL, and BoringSSL to create functional variants. As highlighted in the previous section, the namespace and symbol collision issue that arises from the recombination of the three libraries have successfully been resolved. We were also successful in developing a proof-of-concept application that

can swap out implementations of a function at runtime. We were however unable to resolve the memory error issue which resulted from inaccurate memory allocation by the variants produced. As such, we were unsuccessful in the production of fully functional variants.

While automated recombination methods such as those used by ObjRecombGA [11] might allow for the creation of more functional variants, changes to core data structures such as the SSL context `SSL_CTX` will require the creation of glue code of some kind to account for these differences, either by maintaining parallel versions of the data structures (one for each version of the library used) or by dynamically changing the data structure as it is accessed. While this adaptation could eventually be automated, a first step would be to develop such glue code manually. We have made some progress with the development of such code; finishing this work is a key goal for future work on this project.

One of the surprising results of this work is the degree to which the forks of OpenSSL have diverged. Both BoringSSL and LibreSSL have reorganized the code, changed build systems, and changed core data structures. These changes are likely due to the large technical debt that OpenSSL had incurred over the years which contributed to the large number of vulnerabilities and motivated the creation of the forks in the first place. Indeed, we ourselves ran into issues regarding the build environment of OpenSSL, with us finding that multiple files generated before compilation in OpenSSL are not documented. We can thus understand the motivation for the work that has been done on cleaning up the code and build system in BoringSSL and LibreSSL. Going forward, then, it may be wise to focus more on recombining the forks of OpenSSL rather than using the original library.

The extensive modifications also give additional motivation for the development of FrankenSSL. At a minimum, such extensive code changes have likely introduced vulnerabilities; further, as these codebases diverge, it becomes more and more difficult for vulnerabilities found in any one of them to also be found and removed from the others. Library-level software recombination is a strategy that can mitigate the risks of these vulnerabilities.

VIII. Conclusion

This paper presents a new methodology for defending systems from intrusion through diversity, library-level software recombination. It also presents progress on the development of FrankenSSL, a randomly-generated recombination of code present in OpenSSL, LibreSSL, and BoringSSL.

While we are able to solve the basic problem of doing the recombination using symbol versioning and per-function object file wrappers, changes to core data structures prevents complete interoperability of initialization and deinitialization routines between libraries. Glue code to manage these incompatibilities needs to be developed in order to allow for finer-grained recombination between these libraries. The development of such glue code, automation of the recombination process, and an evaluation of vulnerability mitigation are key goals for future work.

Acknowledgments

The authors would like to acknowledge the support of Canada's Natural Sciences and Engineering Research Council (NSERC) through their Discovery grants program.

References

[1] The PaX Team, "Address space layout randomization," https://pax.grsecurity.net/docs/pax-future.txt, 2003, accessed May 17, 2016.

[2] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2004, pp. 298–307.

[3] J. K. David Evans, Anh Nguyen-Tuong, *Moving Target Defense*. New York: Springer New York, 2011, ch. Effectiveness of Moving Target Defenses.

[4] F. B. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, no. 6, pp. 565–584, 1993.

[5] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier, "Cyberinsecurity: The cost of monopoly," in *Computer and Communications Industry Association Report*, 2003.

[6] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *the Sixth Workshop on Hot Topics in Operating Systems*. IEEE, 1997, pp. 67–72.

[7] 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.

[8] 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.

[9] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *Software Engineering, IEEE Transactions*, vol. SE-12, no. 1, pp. 96–109, 1986.

[10] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

[11] B. Foster and A. Somayaji, "Object-level recombination of commodity applications," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2010, pp. 957–964.

[12] B. C. Foster Jr., "Object file program recombination of existing software programs using genetic algorithms," Master's thesis, Carleton University, 2011.

[13] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS toolkit," https://www.openssl.org/, accessed May 17, 2016.

[14] Google, "BoringSSL," https://boringssl.googlesource.com/boringssl/, accessed May 17, 2016.

[15] OpenBSD, "LibreSSL," http://www.libressl.org/, accessed May 17, 2016.

[16] J. Wagnon, "Security sidebar: LibreSSL is forking OpenSSL," https://devcentral.f5.com/articles/security-sidebar-libressl-is-forking-openssl, May 5, 2014, accessed May 17, 2016.

[17] M. G. Bailey, "Malware resistant networking using system diversity," in *Proceedings of the 6th Conference on Information Technology Education*. ACM, 2005, pp. 191–197.

[18] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2003, pp. 281–289.

[19] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, 2003, pp. 272–280.

[20] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *USENIX Security*, vol. 3, 2003, pp. 105–120.

[21] M. Abadi and J. Planul, "On layout randomization for arrays and functions," in *Principles of Security and Trust*. Springer, 2013, pp. 167–185.

[22] X. Jiang, H. J. Wang, D. Xu, and Y.-M. Wang, "Randsys: Thwarting code injection attacks with system service interface randomization," in *26th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2007, pp. 209–218.

[23] Z. Liang, B. Liang, and L. Li, "A system call randomization based method for countering code-injection attacks," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 1, no. 1, p. 1, 2009.

[24] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *15th USENIX Security Symposium*, 2006.

[25] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," in *Applied Cryptography and Network Security*. Springer, 2004, pp. 292–302.

[26] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to defeat cross-site scripting attacks," *Computers & Security*, vol. 31, no. 4, pp. 612–628, 2012.

[27] S. M. Gustafson, "An analysis of diversity in genetic programming," Ph.D. dissertation, University of Nottingham, 2004.

[28] R. Burke, S. M. Gustafson, and G. Kendall, "A survey and analysis of diversity measures in genetic programming." in *GECCO*, vol. 2, 2002, pp. 716–723.

[29] E. K. Burke, S. Gustafson, and G. Kendall, "Diversity in genetic programming: an analysis of measures and correlation with fitness," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 1, pp. 47–62, Feb 2004.

[30] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 947–954.

[31] OpenSSL Software Foundation, "OpenSSL: Vulnerabilities," https://www.openssl.org/news/vulnerabilities.html, accessed May 17, 2016.

[32] B. Schneier, "Random number bug in debian linux," https://www.schneier.com/blog/archives/2008/05/random_number_b.html, May 19 2008, accessed May 25, 2016.

[33] Codenomicon, "The heartbleed bug," http://heartbleed.com/, April 2014, accessed May 25, 2016.

[34] U. Drepper, "How to write shared libraries," *Retrieved Jul*, vol. 16, p. 2009, 2006.