SoK: Automated Software Diversity

Per Larsen, Andrei Homescu, Stefan Brunthaler, Michael Franz
University of California, Irvine

***Abstract*—The idea of automatic software diversity is at least two decades old. The deficiencies of currently deployed defenses and the transition to online software distribution (the "App store" model) for traditional and mobile computers has revived the interest in automatic software diversity. Consequently, the literature on diversity grew by more than two dozen papers since 2008.**

Diversity offers several unique properties. Unlike other defenses, it introduces uncertainty in the target. Precise knowledge of the target software provides the underpinning for a wide range of attacks. This makes diversity a broad rather than narrowly focused defense mechanism. Second, diversity offers *probabilistic protection* similar to cryptography—attacks may succeed by chance so implementations must offer high entropy. Finally, the design space of diversifying program transformations is large. As a result, researchers have proposed multiple approaches to software diversity that vary with respect to threat models, security, performance, and practicality.

In this paper, we systematically study the state-of-the-art in software diversity and highlight fundamental trade-offs between fully automated approaches. We also point to open areas and unresolved challenges. These include "hybrid solutions", error reporting, patching, and implementation disclosure attacks on diversified software.

I. Motivation

As modern society grows increasingly dependent on the digital domain, adversaries abound in cyberspace. In spite of the combined efforts of the security community, reports of major software vulnerabilities that put millions of users at risk continue to be the norm rather than the exception.

Whereas diversity provides protection and resilience in nature, the commoditization of the computer systems has made them increasingly homogeneous with respect to hardware, operating systems, applications, and everything in between. Homogeneity and standardization provide economies of scale, consistent behavior, and simplify the logistics of distributing programs. We therefore live in a software mono-culture.

Unfortunately, homogeneity has turned out to be a double-edged sword [26]. An attacker can readily download an identical copy of the commodity software running on their victims' systems and probe it for vulnerabilities. After turning a vulnerability into an exploit, the attacker can target all systems running copies of the vulnerable program. In other words, the software mono-culture creates economies of scale for attackers, too.

Artificial software diversity aims to increase the cost to attackers by randomizing implementation aspects of programs. This forces attackers to target each system individually, substantially raising the bar on mass scale exploitation. Without knowledge of the program implementation hosted on a particular system, targeted attacks become significantly harder, too.

The idea of protecting programs with artificially generated diversity is at least two decades old [13]. However, compiler-based software diversity has only recently become practical due to the Internet (enabling distribution of individualized software) and cloud computing (computational power to perform diversification) [25]. These developments and the emergence of code-reuse attacks renewed the interest in software diversity. This has led to a large body of research that is in many ways as diverse as the set of program variants they generate.

This paper systematizes the understanding of software diversity[1] as follows. First, we show the versatility of artificial software diversity by surveying the range of relevant attacks. Second, we provide the first systematic and unified overview of existing diversification approaches. In particular, we characterize the four major properties of a diversification approach using a consistent terminology: (i) what is diversified (Section III), (ii) when and where diversification happens (Section IV), (iii) how security is evaluated (Section V-A), and (iv) the resulting performance overheads (Section V-B). Finally, we point to open areas of research and challenge the belief that compiler-based diversification is less versatile than binary rewriting (Section VI).

II. Today's Security Landscape

Attackers and defenders in cyberspace engage in a continuous arms race. As new attacks appear, new defenses are created in response—leading to increased complexity in both cases. To motivate a study of software diversity, we briefly summarize the evolution and current state of computer security.

A. Taxonomy of Attacks

There is a large spectrum of attacks that an attacker can use against a target, employing a wide range of low-level techniques. We present the ones that are most relevant to automated software diversity.

1) Information Leaks: Often, the attacker seeks to read some sensitive program state he should not have access to. This includes contents of processor registers, memory pages, process meta-data, files, etc. Such information can be valuable to an attacker by itself (e.g., credit card numbers, login credentials, or other personal user information) or to further an on-going attack (by locating protected objects in memory through a pointer leak, or by reading encryption keys used to encrypt other data). In general, information leaks are increasingly used to overcome situations where attackers lack knowledge of program internals [60], [9]. For example, information leaks help bypass address space layout randomization in later stages of an attack.

[1] Research on multi-variant systems overlaps with software diversity. Diversity can be studied in isolation, however. We do so due to space restrictions.

a) Side Channel Attacks: We consider side channel attacks as a category of information leaks. Whereas most other information leaks are intrusive (using some exploit to explicitly reveal program data in unintended ways), side channel attacks infer internal program state in a black box manner by analyzing the interactions between the program and its outside environment. One common measurement used by these attacks is timing; the attacker measures how the time between externally-visible program events changes in response to some stimulus applied by the attacker, and if the response is also correlated with the value of some internal program variable. Many types of stimuli are available to an attacker, such as memory or cache pressure.

2) Memory Corruption Attacks: The attacker often needs to modify the internal program state located in memory. This can be either the end-goal of the attack or an intermediate step (for example, an attacker may seek to modify a function pointer to hijack program control flow). This class covers a large variety of techniques that use programming errors to achieve the same goal: changing the memory contents of the target program.

a) Buffer Overflows: These attacks are a popular instantiation of memory corruption attacks. In a buffer overflow, the attacker writes data outside of the bounds of a memory buffer to corrupt memory adjacent to that buffer. This approach is only viable in languages without bounds checking. This attack is most often applied to stack buffers, but it also works effectively on buffers stored on the program heap.

b) Memory Allocator Exploits: Such exploits rely on the predictability and performance constraints of memory allocators, or on programming errors related to memory allocation, to implement a memory corruption attack. For example, memory allocators typically do not clear unused memory after deallocation requests, because it impacts program performance. This poses significant security problems. Attackers manipulate the allocator in such a way that a newly requested block (under the control of the attacker) overlaps recently released block containing sensitive data. Memory-management errors in applications are also a significant threat; one example are use-after-free errors, where the attacker uses a stale pointer to deallocated memory to read or write a newly allocated block that overlaps the deallocated block.

3) Code Injection: Another way of exploiting a program is to make it execute code under the attacker's control, potentially leading to the attacker taking control of the entire program or even system (programs running with administrator privileges effectively control the entire machine). To achieve this, the attacker injects malicious code into a running program and then redirects execution to the injected code [2]. This attack requires (i) a memory corruption vulnerability, (ii) an executable and writable region of memory, and (iii) a way to direct the processor to execute newly-written data. The third requirement is usually met through memory corruption as well, by modifying a code pointer (like the on-stack return address) to point to the new code. The attacker then crafts a native code payload, writes it to memory, then redirects execution to it. The processor executes the newly inserted block, leaving the attacker in control of the thread of execution.

4) Code Reuse: Operating systems used to allow execution of most program data. This enabled code injection. For example, the entire native stack was executable on Windows and Linux systems. To prevent code injection attacks, operating systems now implement a security model (known as Data Execution Prevention—DEP—or W$\oplus$X [48]) which mandates that a memory page is either writable or executable, but not both at the same time. Code reuse attacks [44], [39], [57], [63] have emerged as a counter-measure to non-executable data defenses. Instead of injecting new code, attackers construct an attack from pieces of executable code (either entire functions [44], [63] or smaller snippets of code) already found in the target program.

Also as a reaction to randomization-based defenses, information leaks have become an increasingly crucial part of code-reuse attacks. One example of this development is a new code-reuse attack called "just-in-time code reuse" [60]. This attack uses information leak techniques to read the code loaded by the target program. Snow et al. avoids page faults by decompiling a page of code at a time and incrementally following references to other mapped pages. After collecting all gadgets, the attack code (containing a built-in gadget compiler) compiles a tailored code reuse payload for that particular program and then runs the attack against the program.

5) JIT Attacks: The introduction of new programming models can change the landscape and introduce new threats to security. In recent years, just-in-time, JIT, compiled languages (such as Java and JavaScript) have become increasingly popular. For example, many dynamically-generated or interactive web pages are written in JavaScript, and all major web browsers contain a JIT compiler for JavaScript. These languages allow programmers to create and run new code dynamically (during program execution); a JIT compiler then translates from source to binary code. This creates a new problem: the attacker can craft and insert malicious source code into the program itself. This is a variant of code injection applied to source code. Program source code is stored as non-executable data, so existing anti-code injection defenses are insufficient. JIT spraying [10] is a recent attack of this kind. When compiling expressions containing constant values, just-in-time compilers may embed the constants directly in binary code. This gives the attacker a way to inject arbitrary binary code into the program, by using constants that contain an attack payload.

6) Program Tampering: The ability to modify a program's state (tamper with the program) has many applications: the attacker can modify unprotected code pointers or instructions to execute arbitrary code, change program data to gain some benefit or bypass DRM protections. One example is bypassing checks in programs that prompt for passwords or serial numbers. Another example of tampering is cheating at computer games, where players give themselves unfair advantages by removing restrictions from the game. Tampering with a program may require the use of one or more of the previously described attacks, as intermediate steps to achieving the desired effect on the target.

Client-side tampering requires unfettered access to the target program, and the attacker is often also in control of the entire physical machine, as well as the operating system, running the program (this model is known as *man-at-the-end*, or MATE [15]). This is a different adversarial model from the other attacks, where the attacker has restricted access to the program, and little or no access to the underlying system.

7) Reverse Engineering: Often, the attacker seeks not to impact the execution of a program, but to find out how the program works internally. Notable uses include reimplementation, compatibility with new software, and defeating security-through-obscurity. Security researchers working on both offenses and defenses use reverse engineering to discover exploitable program vulnerabilities, both from the original program and from patches [22], [18].

B. Taxonomy of Defenses

Many attacks rely on program bugs that are fixable when the program source code is available. If this is not the case, or if attacks rely on intended program behavior, alternative techniques are available. We separate these techniques into the following categories:

1) Enforcement-based Defenses: To defend against attacks, users may opt to take a proactive approach where they seek to prevent attacks from occurring (whenever some program behavior is exploitable, defenders preemptively disallow that behavior). Examples include checking the bounds on array accesses (against buffer overflows), making the data sections non-executable (against code injection) and restricting control flow exclusively to intended paths (to prevent code reuse attacks)—also known as Control Flow Integrity (CFI) [1]. Software Fault Isolation (SFI) [43], [68] is a similar approach that restricts control flow to limited targets within a sandbox. Defenders can deploy these techniques at the source code level, during program compilation [68], or through static binary rewriting [65], [69].

Due to the rise of code injection attacks, modern operating systems (Windows, Linux[2], and OS X) all deploy one low-cost enforcement defense: Data Execution Prevention (DEP, also known as W$\oplus$X). DEP requires the operating system to map all pages containing program data (such as the heap and stack) as non-executable, and all pages of program code as non-writable. This defense has negligible performance costs and effectively stopped most code injection attacks without requiring substantial changes to programs.

Other enforcement techniques require significant changes to protected programs and impose extra restrictions and costs on both programs and programmers. For example, array bounds checking requires extra operations around each array element access; CFI requires extra address checks around each indirect branch. In programs that contain many array accesses or indirect branches, these checks incur significant performance penalties. In addition, programmers have to account for the extra restrictions; for example, they must check that the program does not violate any security restriction during normal program operation. Therefore, we regard this class of defenses as the most intrusive.

2) Program Integrity Monitors: If an attack cannot be prevented, the last line of defense is stopping the program before the attacker has a chance to do any damage. Doing this manually requires significant effort and attention from program users. To stop the program, they first have to notice any unusual behavior in the operation of the program. In many cases, this unusual behavior is either invisible to the user, or is intentionally hidden by the attacker (to prevent detection). While defining when a program is acting "unusually" is very hard, detecting specific attacks is much simpler and can be easily automated in many cases. For each detectable attack, an integrity monitor periodically investigates the state of the running program and checks for signs of an attack.

Examples of such defenses are "stack canaries" [19] and "heap canaries." Code execution attacks often use buffer overflows to overwrite a code pointer, e.g., the return address of the currently executing function. To defend against this attack, modern compilers can insert canaries to guard the return address against changes by pairing it with a randomized guard value—the "canary." Any change to the return address will also change the canary, and the attacker cannot reasonably predict the random value of the canary. On every function return, the program checks the canary against the expected random value and terminates on mismatches. The overheads from the added checks are often negligible (less than 1% on average [61]).

Monitoring defenses are the least intrusive form of defense (in many cases, they can be deployed transparently to the program), but are the most vulnerable to detection and deception. Monitoring allows attackers the same amount of control as long as they remain undetected and they may detect and tamper with the monitor to let the attack succeed.

3) Diversity-based Defenses: Attackers often rely on being able to predict certain details of program implementation, such as the memory locations of sensitive program objects (like code pointers). Removing predictability is, in most cases, as effective as restricting what the attacker can do with the predicted knowledge. Diversification makes program implementations diverge between each computer system or between each execution. This means that the attacker has to either limit the attack to a small subset of predictable targets, or adjust the attack to account for diversity. The latter is impractical in most cases (because it would require duplicated attack effort for each different version of the target), so the malicious effects of the attacks are limited at worst to a small number of targets (where the attacker still gets full control, in absence of any monitoring or enforcement-based defenses). The three following sections treat approaches to diversity in much greater detail. Researchers have investigated the practical uses of automated software diversity against the attacks enumerated in Section II-A (except information leaks and side channels). Figure 1 links attacks to corresponding studies of diversity.

4) Program Obfuscation: Obfuscation to prevent reverse engineering attacks [14], [16] is closely related to diversity and relies on many of the same code transformations. Diversity requires that program implementations are kept private and that implementations differ among systems; this is not required for obfuscation. Pucella and Schneider perform a comparative analysis of obfuscation, diversification, and type systems within a single semantic framework [51].

III. What to Diversify

At the core of any approach to software diversity, whether performed manually by programmers or automatically by a compiler or binary rewriter, is a set of randomizing transformations that make functionally equivalent program copies diverge. A second distinguishing factor among approaches is when

[2]The PaX Team implemented DEP on Linux [48].

diversity is introduced in the software life-cycle. These two choices—*what* to diversify and *when* to diversify—constitute the major axes of the design space and together determine the fundamental properties of any concrete approach. This section focuses on the former choice and Section IV addresses the latter.

Randomizing transformations are conceptually similar to compiler optimizations. Both consist of three steps: (i) determining if a code fragment can be transformed, (ii) estimating if the transformation is profitable, and (iii) applying the transformation. A diversifying transformation differs in the second step by adding an element of chance. The heuristic that determines whether to transform a code fragment or not is replaced (or extended) with a random choice using a pseudo-random number generator (PRNG). Early studies of security via software diversity were compiler-based [13], [24].

Like compiler optimizations, the scope of diversifying transformations varies in granularity from single instructions to the entirety of the program.

5) Instruction Level: These transformations affect at most a few instructions inside a single basic block[3]. Permuting and displacing instructions breaks fine-grained code reuse attacks (assuming implementation details do not leak to attackers [60], [9]). They include but are not limited to:

a) Equivalent Instruction Substitution: The functionality of some instructions overlaps with that of others such that it is often possible to substitute one for another. Load instructions that support multiple addressing modes are common examples.

b) Equivalent Instruction Sequences: Substituting one or more instructions for another instruction sequence leads to even more randomization opportunities. For instance, negation followed by subtraction can substitute for integer addition.

c) Instruction Reordering: It is well known that instructions can execute in any order that preserves the dependencies between data-producing and data-consuming instructions. Using a compiler's instruction scheduler to randomize the instruction order increases diversity among the output binaries.

d) Register Allocation Randomization: While program performance is highly dependent on what variables are allocated to registers, the particular register a variable is assigned to is often irrelevant. Consequently, it is straightforward to randomize register assignments. Register spilling and re-materialization heuristics are amenable to randomization, too.

e) Garbage Code Insertion: This transformation can be as simple as adding no-operation instructions (NOPs), or as complex as inserting entirely new statements. In contrast to other transformations, garbage insertion is always possible and hence allows production of infinitely many program variants.

6) Basic Block Level: The number and ordering of basic blocks within a function or method can be chosen freely. This enables several control-flow transformations including:

[3]A basic block is a sequence of instructions where the execution of the first instruction guarantees that all following instructions are executed, i.e., only the instruction that terminates the block may be a branch.

a) Basic Block Reordering: The last instruction in a basic block can either branch to the successor basic block or have execution fall through to the basic block following it in memory. Reordering makes it necessary to insert additional jumps between pairs of basic blocks chained together on fall-through paths (i.e., without branches) and makes branches in blocks that can fall through after reordering superfluous. Basic block splitting and merging creates additional reordering opportunities.

b) Opaque Predicate Insertion: A single-predecessor block b can be substituted with a conditional branch to b and its clone b' using an arbitrary predicate [14], [16]. These predicates can also guard blocks of garbage code so they never execute.

c) Branch Function Insertion: Branch functions do not return to their callers; instead, they contain code that determines the return address based on the call site [41]. Branch functions can replace direct control transfers via branches and fall-through paths. Similarly, direct calls to functions can be replaced by call functions that obfuscate the call graph.

The first transformation permutes the code layout and breaks fine-grained code reuse attacks. All basic block transformations also complicate the code matching step in patch reverse engineering.

7) Loop Level: Loop-level transformations are suggested by Forrest et al. [24] but not evaluated.

8) Function Level: Transformations at this granularity include:

a) Stack Layout Randomization: Using a buffer overflow to overwrite the return address stored on the machine stack on x86 processors is a classic attack vector. As a result, many randomizing transformations target the stack, including:

- stack frame padding,
- stack variable reordering,
- stack growth reversal, and
- non-contiguous stack allocation.

The last transformation allocates a callee stack frame at a randomly chosen location rather than a location adjacent to the stack frame of the calling function.

b) Function Parameter Randomization: This transformation permutes the existing formal parameters and may add new ones as long as all call-sites can be rewritten to match the actual parameters with the modified formal parameters. This transformation is employed against tampering, code matching and return-into-`libc` attacks.

c) Inlining, Outlining, and Splitting: Inlining the target of a function call into the call-site is a well known compiler optimization. Function outlining is the inverse of inlining: it extracts one or more basic blocks and encapsulates them in their own subroutine. As a special case of function outlining, a function may be split into two; all live variables at the point of the split are passed as parameters to the second function. Together, these transformations randomize the number of function calls and the amount of code duplication among program variants to prevent code matching.

d) Control Flow Flattening: The topology of a function's control-flow graph can be obscured by replacing direct jumps connecting basic blocks with indirect jumps that go through "jump tables." Rather than jumping directly to its successor, each original basic block shares the same successor and predecessor blocks. Again, this complicates code matching.

9) Program Level: Transformations at this level include:

a) Function Reordering: Functions can be laid out in any order within executables and libraries. For dynamically linked functions, the tables maintained by the dynamic linker (e.g., the GOT and PLT dynamic linking structures on Linux systems) can be randomized, too.

b) Base Address Randomization, ASLR: It used to be the case that the base of the code and data segments (i.e. the stack, heap, and statically allocated data) were always loaded at fixed virtual memory addresses. Since the virtual address space of each process is private, the starting address can be chosen at random. Address Space Layout Randomization (ASLR) implements base address randomization and is currently the only deployed probabilistic defense. ASLR complicates memory corruption, code injection, and code reuse attacks, but can be bypassed via information leaks [60], [9].

c) Program Encoding Randomization: It is possible to substitute one encoding of a program for another as long as there is a way to reverse the process. The encoding is reversed by a virtual machine that either interprets the randomized instructions, or emulates a machine for the randomized encoding by translating fragments back to native code prior to execution. Many types of encodings can be used for this purpose. For instance, one of the simplest and fastest encodings computes the exclusive-or of the original program bytes and a randomly chosen key; applying the same transformation when the instructions are about to execute recovers the original encoding. This approach is known as *Instruction Set Randomization* [37], [5]. More complex encodings may compress the instruction stream or offer stronger encryption guarantees. Some encodings are designed to randomize the code layout [29] or code addresses [59]. These transformations can defend against both code injection and fine-grained code reuse attacks.

d) Data Randomization: These transformations aim to stop memory corruption attacks with the exception of constant blinding which defends against JIT-spraying. Several variations are possible:

- *Static Data Randomization.* The layout of static variables can be permuted and padding can be added via dummy variables.
- *Constant Blinding.* A constant c is blinded by applying an injective function $f(c, x) = c'$ where x is a randomly chosen value. During execution, c is obtained by computing $f^{-1}(c', x)$. The exclusive-or operation is a common choice for f and f^{-1}.
- *Structure Layout Randomization.* Composite data structures such as classes and `structs` can have their layout randomized similarly to static data randomization.
- *Heap Layout Randomization.* The layout of dynamically allocated objects can be randomized by adding random padding to each object. The memory allocator can also split the heap into multiple regions and pick a region in which to store each object at random.

e) Library Entry Point Randomization: Library functions are identified by a standardized set of entry points. Each of these entry points can be duplicated and the original entry points can be changed to perform arbitrary functionality, i.e., the `system` in `libc` could be cloned into `system_42` and `system` could be changed to terminate the program. This breaks return-into-`libc` attacks. To function correctly, legitimate programs that use randomized libraries must be updated to use the private set of entry points.

10)System Level: Some transformations are tailored towards system software such as the operating system. *System Call Mapping Randomization*, for instance, is a variant of function parameter diversification that targets the system call interface between processes and the operating system kernel. Without knowledge of the proper system call numbers, the effect of any attack is confined to the compromised process. Applications need to be customized before or after they are installed on the host system to use the correct system call mapping.

Table I gives an overview of the transformations used in the literature. An asterisk next to a checkmark means that the authors presented the transformation without an evaluation. The second column indicates in which stage of the software life-cycle diversification takes place (the stages are: implementation, compilation, linking, installation, loading, execution, and updating). Pre-distribution approaches (marked in Figure 1) have been evaluated with a wider range of transformations—call graph and function parameter randomization, for instance, have not been evaluated with a post-distribution method. The reason, we believe, is that these transformations require interprocedural analysis which is readily supported by compilers but hard to support in binary rewriters. We see that most authors combine at least two randomizing transformations or choose to randomize the program encoding itself.

IV. When to Diversify

The life-cycle of most software follows a similar trajectory: implementation, compilation, linking, installation, loading, executing, and updating. Variations arise because some types of software, typically scripts, are distributed in source form. Figure 1 on page 7 shows how the approaches that we survey fit into the software life-cycle. Some approaches are staged and therefore span multiple life-cycle events; we place these according to the earliest stage. We cover individual diversification techniques according to the software life-cycle from the implementation phase to the update phase.

A diversification engine need not randomize the input program it processes. Several approaches defer diversification by making programs *self-randomizing* [6], [8], [29], [64], [27]. Deferred diversification is typically achieved by instrumenting programs to mutate one or more implementation aspects as the program is loaded by the operating system or as it runs.

Instead of installing several programs instrumented to randomize themselves, the diversification functionality can be included in the operating system [48], [12]. This is exactly how

TABLE I: Overview of randomizing transformations.

Study	When	Transformations																				
		Manual	Equiv. Inst. Subst.	Equiv. Inst. Seq.	Inst. Reordering	Reg. Alloc. Rand.	Garbage Code Ins.	Basic Block Reordering	Opaque Predicate Ins.	Branch/Call Fun. Ins.	Cond. Branch Flipping	Control Flow Flattening	Function Reordering	Function Param. Rand.	Call Graph Rand.	Data Rand.	Syscall. Mapping. Rand.	Base Addr. Rand.	Stack Layout Rand.	Lib. Entry Point Rand.	Prog. Encoding Rand.
Randell [52]	A	✓																				
Avizienis & Chen [11]	A	✓																				
Cohen [13]	C		✓*	✓*	✓*	✓*	✓*								✓*	✓*						✓*
Forrest et al. [24]	C				✓*		✓*	✓*								✓*			✓	✓*		
PaX Team [48]	C, L																	✓				
Chew & Song [12]	C, L																✓		✓	✓		
Bhatkar et al. [6]	I															✓*		✓	✓			
Kc et al. [37]	I																					✓
Barrantes et al. [5]	L, E																					✓
Bhatkar et al. [8]	C												✓			✓		✓	✓			
Kil et al. [38]	B												✓			✓		✓				
Bhatkar et al. [7]	C															✓						
Jakob et al. [36]	I			✓																		
De Sutter et al. [22]	B		✓		✓				✓	✓		✓			✓							
Williams et al. [67]	L, E													✓								✓
Novark et al. [46]	E															✓						
Jackson et al. [35]	C		✓*	✓*		✓*	✓*						✓*			✓*	✓*	✓*	✓*	✓*	✓*	✓*
Wei et al. [66]	E															✓			✓			
Pappas et al. [47]	I		✓		✓	✓																
Hiser et al. [29]	I, E				✓																	✓
Giuffrida et al. [27]	C, E						✓	✓					✓			✓			✓			
Wartell et al. [64]	I, L							✓														
Collberg et al. [15]	C								✓			✓		✓	✓	✓						✓
Shioji et al. [59]	L, E																					✓
Jackson et al. [34]	C						✓															
Homescu et al. [31]	C						✓															
Coppens et al. [18]	U							✓	✓	✓	✓	✓										
Gupta et al. [28]	I												✓									
Davi et al. [21]	L							✓					✓									
Homescu et al. [30]	E						✓									✓						

Legend: A. . .implementation time, C. . .compilation time, L. . .load time, B. . .link time, I. . .installation time, E. . .execution time, U. . .update time.

ASLR[4] is implemented. A compiler prepares the code for base address randomization by generating position-independent code; the operating system loader and dynamic linker then adjust the virtual memory addresses at which the code is loaded.

Consequently, with deferred diversification, all instances of a program share the same on-disk representation—only the in-memory representations vary. This has several important implications. Deferred approaches remain compatible with current software distribution practices; the program delivered to end users by simply copying the program or program installer. When diversification is deferred, the program vendor avoids the cost of diversifying each program copy. Instead the cost is distributed evenly among end users. The end user systems, however, must be sufficiently powerful to run the diversification engine. While this is not an issue for traditional desktop and laptop systems, the situation is less clear in the mobile and embedded spaces.

Second, when diversification is deferred, an attacker does not improve the odds of a successful attack with knowledge of the on-disk program representation.

However, deferred diversification cannot provide protection from certain attacks. Client-side tampering [15] and patch reverse-engineering [17] remain possible since end users can inspect the program binaries before diversification. Software diversification can also be used for watermarking [23]. If a seed value drives the diversification process and a unique seed is used to produce each program variant, the implementation of each variant is unique, too. If each customer is given a unique program variant, and the seed is linked to the purchase, unautorized copying of the program binary can be traced back to the original purchase. However, such use of diversity is also hampered by deferred diversification.

11) Implementation: The idea of software diversity was originally explored as a way to obtain fault-tolerance in mission critical software. Approaches to software fault-tolerance are broadly classified as single-version or multi-version techniques. Early examples of the latter kind include Recovery Blocks [52] and N-Version programming [4] that are based on *design diversity*. The conjecture of design diversity is that components designed and implemented differently, e.g., using separate teams, different programming languages and algorithms, have a very low probability of containing similar errors. When combined with a voting mechanism that selects among the outputs of each component, it is possible to construct a robust system out of faulty components.

Since design diversity is only increased at the expense of additional manpower, these techniques are far too costly to see application outside specialized domains such as aerospace and automotive software. The remaining techniques we will discuss aim to provide increased security, and are fully automatic and thus relevant to a greater range of application domains. This means that diversity is introduced later in the software life-cycle

[4]ASLR is an example of a diversification technique that required compiler customization to produce position independent code. All major C/C++ compilers currently support this security feature.

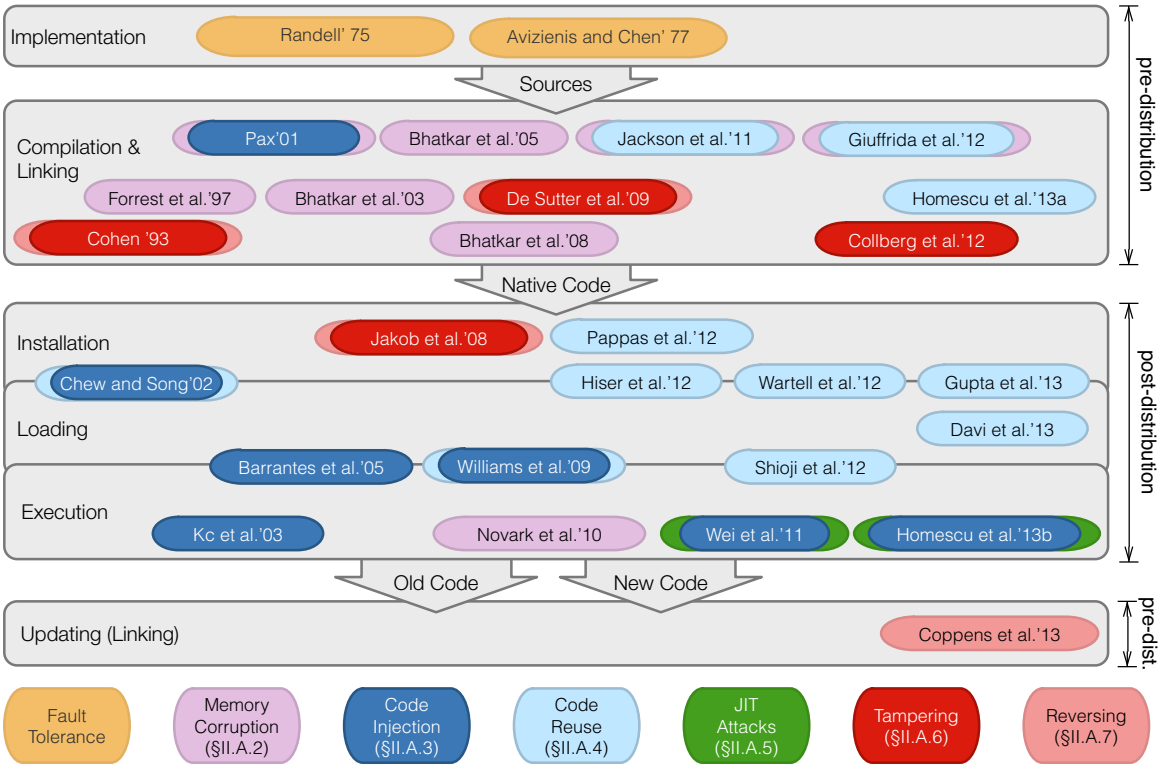

Fig. 1: Approaches to software diversity in relation to the software life-cycle, their inputs, and the attacks they mitigate.

by a compiler or binary rewriting system.

12) Compilation and Linking: Implementing diversity in a compiler makes the process automatic by avoiding changes to the source code of programs. In contrast to humans, compilers do not have a high-level understanding of the input source code and must preserve semantics above all. This limits compile-time diversity (and other fully automated approaches) to program transformations that can be proven to preserve semantics. This generally excludes high-level transformations such as changes to the algorithms that make up a program; as Table I shows, a plethora of lower-level transformations are possible.

Conceptually, compilers are structured as a sequence of code transformation passes. The major passes include parsing the source code into a compiler intermediate representation (IR), performing machine-independent optimizations on the IR, and finally converting the IR into a lower-level representation to perform machine-dependent optimizations. The machine-dependent passes make up the compiler back-end and include optimizations such as instruction selection, instruction scheduling, and register allocation. The randomizing transformations surveyed in the preceding section are often implemented by adding new pipeline passes; randomized register allocation or randomized function inlining only require a few modifications to the heuristics of existing passes. On the other hand, transformations such as garbage insertion or opaque predicate insertion are often added as new compilation passes. When adding a diversifying transformation, care must be taken to prevent later optimization passes from undoing its effects, e.g., dead-code elimination might unintentionally remove garbage code.

From a software engineering perspective, re-purposing a compiler to perform diversification offers at least four benefits:

Reuse avoids duplication of effort: Many of the randomizing transformations described in Section III require data-flow analysis to determine if the randomizing transformation is possible. Since compilers already contain the prerequisite analyses [62], such transformations are easy to support.

Compilers target multiple hardware platforms: Compilers are highly sophisticated and thus costly to produce and maintain. The high costs are typically amortized by supporting multiple instruction sets in the compiler back-end. The GNU Compiler Collection release 4.8, for example, supports over 50 different hardware models and configurations. Consequently, randomizing transformations can easily target all platforms supported by the host compiler.

Compilation avoids the need for disassembly: The transformation from source code to object code is a lossy transformation. Optimizations obscure the original program structure and code and data is interspersed. As a result, perfect recovery of the original program control flow is not generally possible [13], [33]. Consequently, disassemblers rely on heuristics that work most of the time. To preserve correctness when these heuristics fail, runtime mechanisms are necessary to detect and recover from disassembly errors.

Compilers support profile guided optimization: It is well known that many programs spend roughly 90% of their time executing 10% of the code. Most compilers can instrument and execute a program to discover the "hot" code paths. A subsequent compilation pass uses the profile information to better optimize frequently executed code at the expense of less frequently executed code [50]. Since diversification tends to make programs run slower, reducing the amount of diversification for hot code fragments significantly lowers the performance overhead [17], [31].

Unfortunately, it is not always possible to customize a compiler. While two of the major production compilers in use today—the GNU Compiler Collection and LLVM—have open source licenses, several proprietary compilers remain in widespread use. Absent any extension mechanism and vendor support, proprietary compilers cannot act as diversification engines. While customizing a compiler may be the natural way to implement diversification, two alternatives are also available. First, source-to-source transformations can be applied *prior* to compilation. Bhatkar et al. do so to make programs self-randomizing at load time [8]. Second, program diversification can happen *after* compilation in one of two ways. The first way is to instruct the compiler to output assembly code such that it can be rewritten before it is assembled and linked [40]. The second way is to disassemble and rewrite the object files produced by the compiler before or during linking [6]. These, link-time diversification techniques have the following advantages:

Debugging information is available: Software vendors typically strip binaries of debug information before they are distributed since debugging information facilitates reverse engineering. So, in contrast to post-link diversification, reliable static disassembly is feasible.

The approach is compatible with proprietary compilers and linkers: Diversification after compilation (and before linking) is possible even on platforms where neither the compiler nor the linker is amenable to customization.

Whole program diversification is possible: Compilers typically process one translation unit, i.e., a source file and the headers it includes, at a time. This gives compilers a limited view of the program, meaning that certain transformations are not possible. Function reordering, for instance is not practical before all functions have been compiled, i.e., at link time.

Pre-distribution approaches (that do not produce self-randomizing binaries) generally share two drawbacks in contrast to the post-distribution techniques we cover later in this section:

Cost of producing program variants: If programs are diversified before they are distributed to end users, software vendors must purchase the computational resources to generate a program variant for each user. At first, it may seem that if it takes n minutes to compile a program, generating a unique variant for x users takes $n*x$ time which obviously is expensive for popular and complex software. However, Larsen et al. [40] show that much of the work to create each variant is repetitive and can be cached to reduce compilation time by up to 92%.

Increased Distribution Costs: While pre-distribution methods ensure that clients cannot disable diversification, each client must download a separate program variant. This requires changes to the current software distribution channels. Rather than cloning a "golden master" copy, distribution systems must maintain a sufficiently large inventory of program variants such that downloads start without delay. Not all inventory may be used before new program versions are released. These changes will most likely also affect the content distribution networks used for high volume software.

Note that ahead-of-time compiled languages incur both of these costs while just-in-time compiled languages, such as Java and JavaScript, do not (compilation to machine code happens on the clients).

13) Installation: We now move to approaches where diversification happens during or after program installation on the host system and before it is loaded by the operating system.

The need to disassemble stripped binaries is a major challenge at this stage. As previously mentioned, error-free disassembly without debugging symbols is not generally possible. The compiler intersperses code and data, i.e., by inserting padding between functions and embedding jump tables, constant pools, and program meta-data directly in the instruction stream.

Post-installation and load-time diversification must disassemble program binaries before they run. Typically, a powerful, recursive disassembler such as Hex-Rays IDA Pro is used for this process. A recursive disassembler uses a worklist algorithm to discover code fragments inside a binary. The worklist is initially populated with the program entry point(s), additional code fragments are put on the list by discovering control flow edges by analyzing the calls and branches of each list item. Unfortunately, the problem of determining whether the control flow can reach a particular code location is equivalent to the halting problem and thus undecidable [13, p. 578]. Disassemblers therefore err on the side of not discovering all code [47] or, alternatively, treat all of the code section as instructions even though some bytes are not [29], [65].

In-place diversification is an install-time-only approach that sidesteps the problem of undiscovered control flow [47]. Code sequences reachable from the program entry point are rewritten with other sequences of equal length. Unreachable bytes are left unchanged thus ensuring that the topology of the rewritten binary matches that of its original. In-place rewriting preserves the addresses of every direct and indirect branch target and thereby avoids the need for and cost of runtime checks and dynamic adjustment of branch targets. The approach does have two downsides, however: (i) undiscovered code is not rewritten and thus remains available to attackers and (ii) preserving the topology means that return-into-`libc` attacks are not thwarted.

Most other post-installation diversification approaches are staged and include actions at multiple steps in the application life cycle. Typically a program is prepared for randomization after it has been installed and is randomized as it is loaded.

Instruction location randomization (ILR) rewrites binaries to use a new program encoding [29]. ILR changes the assumption that, absent any branches, instructions that are laid out sequentially are executed in sequence; instructions are instead relocated to random addresses by disassembling and rewriting programs as they are installed on a host system. A data

structure, the *fallthrough* map, contains a set of rewrite rules that map unrandomized instruction locations to randomized ones and to map each randomized instruction location to its successor location. To avoid the need to separate code and data, rewrite rules are generated for all addresses in a program's code section. A process virtual machine, Strata [55], executes the rewritten programs. At runtime, Strata uses the fallthrough map to guide instruction fetch and reassemble code fragments before execution; fragments are cached to reduce the translation overhead.

Binary stirring [64] is also a hybrid approach that disassembles and rewrites binaries as they are installed such that they randomize their own code layout at load time. Rather than using a process virtual machine, randomization is done via a runtime randomizer that is unloaded from the process right before the main application runs. This ensures that the code layout varies from one run to another. Instead of trying to separate data from code, the text segment of the original binary is treated as data and code simultaneously by duplicating it into two memory regions—one which is executable and immutable and one which is non-executable but mutable. One duplicate is left unmodified at its original location and marked non-executable. The other duplicate is randomized and marked executable. Reads of the text segment go to the unmodified duplicate whereas only code in the randomized duplicate is ever executed. All possible indirect control flow targets in the unmodified duplicate are marked via a special byte. A check before each indirect branch in the randomized duplicate checks if the target address is in the unmodified duplicate and redirects execution to the corresponding instruction in the randomized duplicate.

Chew and Song customize each operating system installation to use randomized system call mappings and randomized library entry points [12]. Application programs must therefore be customized to the system environment before they run. Since the goal of rewriting the binaries is to use the correct system call identifiers and library entry points of a particular system, the topology of the binaries does not change. Again, this means that undiscovered control flow is not an issue. However, the problem of undiscovered code implies that rewriting may fail to update all system and library calls.

The drawbacks of static binary rewriting can be summarized as follows:

Overheads of runtime checking: Most static rewriting solutions include a runtime mechanism to compensate for static disassembly errors. For instance, undiscovered control flows to addresses in the original program may be dynamically redirected to the corresponding locations in the rewritten program—e.g., by placing trampoline code at the addresses containing indirect branch targets in the original program. The compensation code invariably adds an overhead to the rewritten program, even without diversification, because it increases the working set and instructions executed relative to the original program. Some static binary rewriters omit these compensation mechanisms [47], [21]. Such approaches are unsafe; program semantics may not be preserved due to disassembly errors.

Incompatibility with code signing: Commercial binaries use digital signatures and modern app stores require them. This allows the operating system to establish the provenance of the code and verify its integrity before launching an application. Binary rewriters that change the on-disk representation of programs cause these checks to fail.

Heterogeneity of binary program representations: Program binaries do not solely consist of machine code; they also contain various forms of meta-data such as relocation information, dynamic linker structures, exception handling meta-data, debug information, etc. Static binary rewriters must be able to parse this meta-data to discover additional control flow. The format of this meta-data is not only operating system specific—it is also specific to the compiler and linker that generated the binary. So in contrast to compilers, whose input languages are mostly platform agnostic, it requires far more effort to support multiple operating systems and compilers in a binary rewriter.

All post-distribution approaches, e.g., those that diversify software on the end user's system rather than prior to its distribution, share several key advantages and drawbacks. The advantages are:

Legacy binaries without source code can be diversified: Since these approaches require no code-producer cooperation, legacy and proprietary software can be diversified without access to the source code.

Distribution of a single binary: Post-distribution diversification remains compatible with the current practice of distributing identical binaries to all users.

Amortization of diversification costs: Unlike pre-distribution techniques, post-distribution diversification spreads this cost among the entire user base.

The drawbacks are:

No protection against client side attacks: Since post-distribution diversification runs on clients, the process can be disabled by malware or the end users themselves. If diversity is used to watermark binaries and raise the cost of reverse engineering and tampering, it must be applied prior to distribution.

The diversification engine increases the trusted computing base: Widely distributed software such as the Java Virtual Machine, Adobe Reader and Flash are valuable to attackers. Since all systems must host a copy of the diversification engine, it becomes another high visibility target.

No support for operating system diversification: In contrast, several compile-time diversification approaches support operating system protection [12], [27], [34]. Rewriting kernel code is not impossible but it is rather involved because kernels differ from application code in numerous ways. Kernel code is self-loading and does not adhere to a particular binary format; Linux images, for instance, consist of a small decompression stub and a compressed data-stream. The control flow in kernels is also particularly hard to analyze due to extensive use of hand-written assembly and indirect function calls for modularity. The control flows from system calls, exception handlers and interrupt handlers are implicit and must be discovered by parsing kernel specific data structures. Additionally, some code cannot be altered or moved since it interacts closely with the underlying hardware.

14) Loading: Load-time diversification approaches do not change the on-disk representation of programs. Rather, they perform randomization as these are loaded into memory by the operating system.

Many deferred diversification approaches perform randomization at load-time. With ASLR for instance, the compiler prepares binaries for randomization during code generation while the loader selects the randomized base address.

Several load-time diversification approaches also have runtime components. Barrantes et al. [5], for instance, randomizes the instruction set encoding as code is loaded and uses the Valgrind dynamic binary rewriter [45] to decode the original machine instructions as they are about to execute. Williams et al. [67] similarly implement instruction set randomization atop the Strata process virtual machine. Their approach even has a compile-time component to prepare binaries by adding an extra "hidden" parameter to each function. This parameter acts as a per-function key whose expected value is randomly chosen at load-time and checked on each function invocation. The process virtual machine instruments each function to verify that the correct key was supplied; it also randomizes the instruction set encoding to prevent code injection. Without knowledge of the random key value, function-level code-reuse (e.g., return-into-`libc`) attacks are defeated. Finally, Shioji et al. [59] implement address-randomization atop the Pin [42] dynamic binary rewriter. A checksum is added to certain bits in each address used in control flow transfers and the checksum is checked prior to each such transfer. Attacker injected addresses can be detected due to invalid checksums. Checksums are computed by adding a random value to a subset of the bits in the original address and hashing it.

In contrast to these hybrid approaches, Davi et al. [21] implement a pure load-time diversification approach that randomizes all segments of program binaries. The code is disassembled and split into code fragments at call and branch instructions. The resulting code fragments are used to permute the in-memory layout of the program. The authors assume that binaries contain relocation information to facilitate disassembly and consequently omit a mechanism to compensate for disassembly errors.

In addition to the general benefits of post-distribution, the particular benefits of load-time diversity are:

Compatibility with signed binaries: Load-time diversification avoids making changes to the on-disk representation of binaries and therefore permits integrity checking of signed binaries in contrast to post-installation rewriting approaches.

Dynamic disassembly: With the exception of Davi et al. [21], load-time approaches are based on dynamic binary rewriting. Rather than trying to recover the complete control flow before execution, the rewriting proceeds on a by-need basis starting from the program entry point. Control flow transfers to code that has not already been processed are intercepted and rewritten before execution; already translated code fragments are stored in a code cache to avoid repeated translation of frequently executed code. This avoids disassembly errors and consequently the need to handle these at runtime.

The drawbacks of load-time approaches are:

Runtime overhead of dynamic rewriting: Dynamic rewriting, like dynamic compilation, happens at runtime and thereby adds to the execution time. In addition, the binary rewriter itself, its meta-data, and code cache increase the pressure on the cache hierarchy and the branch predictors.

No sharing of code pages for randomized libraries: Operating systems use virtual memory translation to share a single copy of a shared library when it is loaded by multiple processes. Since libraries such as `libc` are loaded by almost every process on a Unix system, this leads to substantial savings. However, load-time rewriting of shared libraries causes these to diverge among processes which prevents sharing of code pages.

15) Execution: The preceding sections have already covered approaches with runtime aspects, e.g., those involving dynamic binary rewriting. We now focus on diversification that primarily takes place during execution. The fact that certain techniques, i.e., dynamic memory allocation and dynamic compilation, cannot be randomized before the program runs motivates these approaches. Consequently, runtime diversification approaches complement all previously discussed approaches by randomizing additional program aspects.

Many heap-based exploits rely on the heap layout. Randomizing the placement of dynamically allocated data and meta-data makes such attacks more difficult. The heap layout is randomized one object at a time by modifying the memory allocator [46]. The diversifying allocator has several degrees of freedom. It can lay out objects sparsely and randomly in the virtual address space rather than packing them closely together. It can also fill objects with random data once they are released to neutralize use-after-free bugs.

Dynamic code generation has also been exploited via JIT-spraying attacks against web browsers that compile JavaScript to native code. Like ahead-of-time compilers, just-in-time compilers can be modified to randomize the code they generate [66]. For legacy and proprietary JIT-compilers, dynamic binary rewriting enables randomization without any source code changes at the expense of a higher performance penalty [30]. Such rewriting, however, creates higher overheads than code randomization done directly by the JIT-compiler.

16) Updating: Program patches are the delivery vehicle for security and usability improvements. Attackers can compute the code changes between two versions by comparing a program before and after applying an update. Unfortunately, knowledge of the code changes helps adversaries locate exploitable bugs and target users that have not yet updated their software.

Software updates can be protected by diversifying each program release before generating the patch. This has two beneficial effects. First, diversification will make the machine code diverge even in places where the source code of the two program releases do not; this potentially hides the "real" changes in a sea of artificial ones. Second, diversification can be done iteratively until the heuristics used to correlate two program versions fail; this greatly increases the required effort to compare two program releases at the binary level [18], [17]. Note that diversity against reverse engineering program updates works by randomizing different program releases (temporal diversity) rather than randomizing program implementations between different systems (spatial diversity).

Software diversity can also be used to introduce artificial software updates. It is generally recognized that, given full access to a binary, a determined adversary can successfully reverse engineer and modify the program given enough time [13]. If the binary requires server access to work, the server can force client systems to download a continuous stream of updates by refusing to work with older client software releases [15]. This exhausts the resources of an adversary since he is forced to reverse engineer each individual release.

V. Quantifying the Impact of Diversity

The preceding section described software diversification approaches in qualitative terms. While this is important to *implementors* of diversification engines, it does not help *adopters* of diversified software quantify the associated costs and benefits. This section survey how researchers quantify the security and performance impacts of software diversity.

A. Security Impact

Software diversity is a broad defense against current and future implementation-dependent attacks. This makes it hard to accurately determine its security properties before it is deployed. The goal of diversity—to drive up costs to attackers—cannot be measured directly, so researchers resort to proxy measurements. Ordered from abstract to concrete, the security evaluation approaches used in the literature are:

Entropy analysis: Entropy is a generic way to measure how unpredictable the implementation of a binary is after diversification. Low entropy solutions, e.g., ASLR on 32-bit systems, are insecure because an attacker can defeat randomization via brute-force attacks [58]. Entropy, however, overestimates the security impact somewhat since two program variants can differ at the implementation level and yet be vulnerable to the same attack.

Attack-specific code analysis: The construction of certain attacks has been partially automated. Gadget scanners [53], [32], [54], for instance, automate the construction of ROP chains. These tools are typically used to show that a code reuse attack generated by scanning an undiversified binary stops working after diversification. However, adversaries could collect a set of diversified binaries and compute their shared attack surface which consists of the gadgets that survive diversification—i.e., they reside at the same location and are functionally equivalent. Homescu et al. [31] use this stricter criterion—surviving gadgets—in their security evaluation.

Logical argument: Early papers on diversity did not qualify the security properties and rely on logical argumentation instead [13]. For instance, if an attack depends on a particular property (say the memory or code layout of an application) which is randomized by design, then the defense must succeed. Unfortunately, such reasoning does not demonstrate the entropy of the solution, i.e., how hard it is for an attacker to guess how a program was randomized.

Testing against concrete attacks: Often, researchers can build or obtain concrete attacks of the type their technique defend against. Showing that such attacks succeed before diversification but fail afterwards is a common proxy for security. Again, such testing does not imply high entropy.

TABLE II: Security impact of transformations.

Study	Defends Against	Evaluation			
		Entropy	Specific	Logical	Attack
Cohen'93 [13]	Many			✓	
Forrest et al.'97 [24]	Stack Buf. Ovf.			✓	✓
PaX Team'01 [48]	Mem. Corr., Code Inj.			✓	
Chew & Song'02 [12]	Buffer Ovf			✓	
Bhatkar et al.'03 [6]	Many	✓		✓	
Kc et al.'03 [37]	Code Inj.	✓		✓	
Barrantes et al.'05 [5]	Code Inj.	✓	✓	✓	✓
Bhatkar et al.'05 [8]	Many	✓		✓	
Kil et al.'06 [38]	Mem. Corr.	✓	✓	✓	
Bhatkar et al.'08 [7]	Mem. Corr.	✓		✓	
De Sutter et al.'09 [22]	Code Matching		✓	✓	
Williams et al.'09 [67]	Code Inj., Code Reuse	✓		✓	✓
Novark et al.'10 [46]	Mem. Alloc., Heap Buf. Ovf.	✓		✓	
Jackson et al.'11 [35]	Many			✓	
Wei et al.'11 [66]	Heap Spray, JIT Spray	✓		✓	
Pappas et al.'12 [47]	Code Reuse	✓	✓	✓	✓
Hiser et al.'12 [29]	Code Reuse	✓	✓	✓	✓
Giuffrida et al.'12 [27]	Many	✓		✓	
Wartell et al.'12 [64]	Code Reuse	✓	✓	✓	✓
Collberg et al.'12 [15]	Tampering	✓	✓	✓	✓
Shioji et al.'12 [59]	Code Reuse	✓		✓	✓
Jackson et al.'13 [34]	Code Reuse			✓	
Homescu et al.'13a [31]	Code Reuse		✓	✓	✓
Coppens et al.'13 [18]	Code Matching		✓	✓	✓
Gupta et al.'13 [28]	Code Reuse	✓		✓	✓
Davi et al.'13 [21]	Code Reuse	✓	✓	✓	✓
Homescu et al.'13b [30]	JIT Spray, Code Reuse			✓	

Since each of the ways to evaluate security impacts are imperfect, authors often use both abstract and concrete security evaluations. Table II shows how each implementation evaluates the impact of their approach. One commonality among all evaluations is the assumption that the effects of diversification remain hidden from attackers. However, in Section VI-C we highlight vulnerabilities that enable implementation disclosure and thereby undermine this assumption.

B. Performance Impact

The chance that a security technique sees adoption is arguably inversely proportional to its performance overhead. So far, the only ones that have been widely adopted (ASLR, DEP, and stack canaries) are those with negligible performance impact. For another technique to be adopted at large, its performance impact must be below 5-10% according to Szekeres et al. [61].

Different studies of diversity measure performance cost differently. The most popular benchmark for this is the SPEC CPU benchmark suite, usually the most recent version available (at present, that is SPEC CPU 2006). In cases where SPEC CPU is not available or appropriate as a benchmark, implementations measure the CPU impact on other workloads, such as real-world applications (Apache, Linux command line utilities, the Wine test suite) or other CPU benchmarks. As the implementations of most of the techniques we discussed are not publically available, we rely on self-reported performance numbers from their

TABLE III: Costs of transformations.

Study	Stage	Benchmark	Performance Overhead	Code Increase	Memory Increase	Language
Randell'75 [52]	Impl.	N/A				
Avizienis & Chen'77 [11]	Impl.	N/A				
Cohen'93 [13]	Comp.	N/A				
Forrest et al.'97 [24]	Comp.	N/A				
PaX Team'01 [48] (ASLR results by Payer [49])	Comp., Load	SPEC CPU 2006 32-bit SPEC CPU 2006 64-bit	9% 2%	N/A N/A		(Language Agnostic)
Chew & Song'02 [12]	Comp., Load	N/A				C/C++
Bhatkar et al.'03 [6]	Inst.	Linux utils	0%-21%	N/A		C/C++
Kc et al.'03 [37]	Inst.	ftp sendmail fibonacci	33% 1974% 28781%	N/A		C/C++
Barrantes et al.'05 [5]	Load, Exec.	Apache—SPEC web 99	62%	N/A		C/C++
Bhatkar et al.'05 [8]	Comp.	Linux utils, Apache	11%	N/A		C/C++
Kil et al.'06 [38]	Link	SPEC CPU 2000 LMBench Apache	0% 3.57% 0%	N/A		C/C++
Bhatkar et al.'08 [7]	Comp.	Linux utils	15%	N/A		C/C++
De Sutter et al.'09 [22]	Link	SPEC CPU 2006	5%-10%	N/A		C/C++
Williams et al.'09 [67]	Load, Exec.	SPEC CPU 2000—ISR SPEC CPU 2000—CSD	17% 54%	N/A		C/C++
Novark et al.'10 [46]	Exec.	SPEC int 2006 Firefox	20% 5%	N/A		C/C++
Jackson et al.'11 [35]	Comp.	N/A				C/C++
Wei et al.'11 [66]	Exec.	V8	5%	N/A		JavaScript
Pappas et al.'12 [47]	Inst.	Wine tests	0%	N/A		C/C++
Hiser et al.'12 [29]	Inst., Exec.	SPEC CPU 2006	13%-16%	14MB-264MB	14MB-345MB	C/C++
Giuffrida et al.'12 [27]	Comp., Exec.	SPEC CPU 2006 devtools	4.8% 1.6%	N/A		C/C++
Wartell et al.'12 [64]	Inst., Load	SPEC CPU 2000 Linux coreutils	4.6% 0.3%	73%	37%	C/C++
Collberg et al.'12 [15]	Comp.	SPEC CPU 2000	5%-10%	N/A		C/C++
Shioji et al.'12 [59]	Load., Exec.	bzip2	265%-2510%	N/A		C/C++
Jackson et al.'13 [34]	Comp.	SPEC CPU 2006 Apache	5%-10% 11.3%	20%-50%	N/A	C/C++
Homescu et al.'13a [31]	Comp.	SPEC CPU 2006	1%	N/A		C/C++
Coppens et al.'13 [18]	Upd.	SPEC CPU 2000	5%-30%	15%-20%	5%-30%	C/C++
Gupta et al.'13 [28]	Inst.	N/A				C/C++
Davi et al.'13 [21]	Load	SPEC CPU 2006	1.2%-5%	1.76%	5%	C/C++
Homescu et al.'13b [30]	Exec.	V8 HotSpot	250% 15%	N/A		JavaScript Java

authors. Along with the average impact of each implementation on program running time, we also show the effects on memory usage and on-disk binary file size (when reported). Table III shows the time and space cost of each technique.

For pre-distribution approaches, the overheads generally range from 1 to 11%. For post-distribution methods, the range of reported overheads is greater and typically range from 1% to 250% indicating that implementations of these approaches must take greater care to keep overheads in check. (Note that Pappas et al. [47] use an unorthodox benchmarking approach and that we consider the approaches by Kc et al. [37] and Shioji [59] to be outliers.) While the benchmarking methodology varies considerably, we conclude that both pre and post-distribution approaches can result in low runtime overheads [31], [21]. We also see greater variability in the binary size overheads among post-distribution approaches when compared to pre-distribution approaches; in both cases, the overheads are small to moderate. Some post-distribution approaches also increase runtime memory overheads—between 5% and 37%.

Note that Table III excludes ahead-of-time costs associated with diversification. For pre-distribution methods, the software developer or distributor may pay the diversification costs. For post-distribution methods, end users contribute the computing resources to diversify programs during installation, loading or running. It remains to be seen if on-device diversification is practical on resource and power-constrained computers such as mobile devices and embedded systems.

VI. Open Areas and Unsolved Challenges

While the security and performance implications of diversified software are well understood, several practical concerns remain to be addressed. In addition, existing research has not fully explored the protective qualities of diversified software nor has it reached consensus on how to evaluate the efficacy of software diversity with respect to the attacker workload. For example, we think that it can provide probabilistic protection against the long standing problem of covert channels.

A. Hybrid Approaches

A schism exists between proponents of compilation-based diversification and diversification via binary rewriting. It is frequently argued that binary rewriting is preferable to compiler-based methods because the latter require source code access, custom compilers, and require changes to current program distribution mechanisms [67], [64], [29], [47]. However, binary rewriting approaches are inherently client-side solutions and therefore cannot defend against tampering or discourage piracy via watermarking [40]. Moreover, a decompiler that

produces LLVM compiler intermediate representation [3] can be combined with a compiler-based diversification engine. This results in a hybrid approach where the same randomizing transformations can be applied to source code as well as legacy binaries.

Another hybrid-approach of interest is centered around compiler-rewriter cooperation. Static binary rewriting of stripped binaries suffers from incomplete information. The code-data separation problem could be entirely avoided if the compiler (or linker) contains a map of all indirect branch targets; this information is readily available at compile and link-time. Enabling reliable disassembly not only simplifies the implementation of binary rewriters and improves their throughput, but the resulting binaries also run faster without the need to detect and correct disassembly errors at runtime.

B. Error Reports and Patches

Current best practices dictate that program crashes on end-user systems can be reported back to the software developers. The developers use these reports to prioritize and address bugs and to produce software updates that improve the stability and security of their products.

Error reports contain machine state information such as the instruction pointer plus stack and register contents at the time of the crash. The reports are sent to a server that performs two tasks: it uses debug information to determine the source code location of the crash and it matches the new error report with previous reports to rank bugs by frequency.

Unfortunately, software diversity interferes with the processing of error reports. The randomization of the program implementation makes error reports diverge even if two users trigger the exact same error. Programs are typically distributed without debugging information and therefore report crashes using the instruction pointer plus the register and stack contents. Developers store a single copy of the debugging information for each software release to translate locations in the binary into source code locations in a process known as symbolication. With code layout randomization, however, the instruction pointer corresponding to a particular source code line will vary between variants. If not addressed, diversification interferes with symbolication of error reports.

A straw-man solution is to generate or store debug information for each program variant on the error reporting server. Unfortunately, this is space consuming and impractical for client-side diversification approaches. The alternative is to hide the effects of diversification from error reporting frameworks. Error reports can be transformed to a "canonical" version matching what an undiversified copy of the program would report for the same error. This requires a way to integrate with existing error reporting mechanisms and meta-data to drive the transformations.

Diversification approaches that randomize the on-disk representation of programs also interfere with software patches. If each user has a unique program copy, patches must be customized to each individual copy. Neither of these challenges have received much attention to date.

C. Implementation Disclosure

Besides low entropy, information leakage threatens the effectiveness of diversified defenses. Information leaks are accidental disclosures of the layout or contents of process or kernel memory [56]. ASLR shifts all addresses by the same amount such that relative distances within a library remain unchanged; this means that an attacker can infer the entire code layout if a single code address is disclosed.

Finer-grained code randomization also affects the relative distances between code fragments. Bypassing such randomization requires disclosure of multiple code addresses. Snow et al. [60] demonstrate a *just-in-time* code-reuse attack that uses JavaScript to discover the code layout (via a bounds checking error in C++ code) and to build a customized ROP chain, thus defeating fine-grained code-diversification. Bittau et al. [9] perform a "blind return-oriented-programming" attack by exploiting a buffer overflow in the `nginx` web-server and using the response (crash, no crash) as a side channel to incrementally guess the position of a required gadget set in fully diversified binaries. These attacks call for work on new types of diversity that prevent or tolerate (partial) information disclosure.

Crane et al. [20] propose that diversified binaries be "booby-trapped" with code that detects guessing attacks. A booby-trap is an instruction sequence beginning with an unconditional branch past its last instruction so the trap is skipped during normal execution; attempts to execute code at random addresses, however, will eventually trigger the trap. Traps help alert defenders to attacks and may even allow programs to operate through attacks by recovering corrupted state.

D. Measuring Efficacy

The study of how diversity affects the adversary's effort is in its infancy. Many of the works we survey report several detailed performance metrics on standardized benchmarks. This is in stark contrast to the security evaluations which are frequently qualitative, i.e., based on a logical argument on why attacks fail, an analytical calculation of the resulting entropy, or the demonstration of concrete attacks that fail.

Few studies quantify the impact of diversification—e.g., by counting surviving gadgets when defending against ROP attacks [31] or the percentage of code that can be matched after diversification [17]. On a similar note, entropy results determine the space of a diversified population, but there are no studies that show how well program variants are distributed within that space. This matters because some transformations are not always legal; performing equivalent instruction selection on a program with thousands of functions can generate many variants in principle, but if only a handful of functions use instructions that can be substituted, then the resulting population is not sufficiently diverse.

We believe the reasons for this are twofold. First, there is a lack of consensus on acceptable methodologies. Second, having a set of publicly available tools to evaluate and compare approaches with would reduce the effort to evaluate security. Numerous papers have been published on how to perform sound performance evaluations; we think a similar effort should be undertaken with respect to efficacy metrics for diversified software.

E. Diversity as a Counter to Side Channel Attacks

Covert channels exist whenever computation has observable side-effects. For example, the time to divide two numbers depends on the operands and the time to access a memory location depends on the state of the memory hierarchy.

Attackers can build statistical models that correlate sensitive program data, e.g., cryptographic keys, with side-effects by observing the target program operating on known inputs. The growing popularity of cloud computing, which co-locates computation of its customers on shared hardware, only increases the need for a solution to this long standing problem.

One of the key requirements to build a side-channel attack is the ability to accurately replicate the victim environment. However, software diversity breaks exactly this assumption. Attackers no longer have easy access to an exact copy of the target program. Consequently, we expect that existing and side-channel specific randomizing transformations provides an effective counter to this long standing threat.

VII. Conclusions

The overall idea of software diversity is simple. However, we show how the interactions with current development, distribution, and security practices are not. We bring clarity to this complex field by treating the major axes of software diversity independently while using a consistent terminology.

There is a tension between pre-distribution and post-distribution diversification approaches. The former are easy to implement portably, support the widest range of transformations, and can defend against client-side attacks. The latter support legacy and proprietary software, amortize diversification costs, and require no changes to current distribution mechanisms. In terms of performance, approaches of both kinds can deliver acceptable overheads (as low as 1-5%). The two fastest binary rewriters may not preserve program semantics [47], [21] though. With two exceptions [21], [40] research in software diversity does not consider compatibility with security features such as crash reporting and code signing.

Naturally, the research in software diversity can be extended; we point out several promising directions. There is currently a lack of research on hybrid approaches combining aspects of compilation and binary rewriting to address practical challenges of current techniques. We also point to the need to address memory disclosure attacks on diversity and finally argue that diversified software may provide an effective defense against side channel attacks.

Acknowledgments

We thank the anonymous reviewers, Prof. Greg Morrisett, Stephen Crane, and Mark Murphy for their insightful reviews, helpful suggestions, and proofreading.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, 2005.

[2] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49), 1996. http://www.phrack.org/issues.html?id=14&issue=49.

[3] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, 2013.

[4] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the International Computer Software and Applications Conference*, COMPSAC '77, pages 149–155, 1977.

[5] E. Barrantes, D. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[6] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, SEC '03, pages 105–120, 2003.

[7] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, 2008.

[8] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, SEC '05, pages 271–286, 2005.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P '14, 2014.

[10] D. Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX Workshop on Offensive technologies*, WOOT'10, 2010.

[11] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*, FTCS '95, page 113, 1995.

[12] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, 2002.

[13] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[14] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, 1997.

[15] C. S. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, 2012.

[16] C. S. Collberg, C. D. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, 1998.

[17] B. Coppens, B. De Sutter, and J. Maebe. Feedback-driven binary code diversification. *Transactions on Architecture and Code Optimization*, 9(4), Jan. 2013.

[18] B. Coppens, B. D. Sutter, and K. D. Bosschere. Protecting your software updates. *IEEE Security & Privacy*, 11(2):47–54, 2013.

[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, D. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, SEC '98, pages 63–78, 1998.

[20] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *Proceedings of the 2013 Workshop on New Security Paradigms*, NSPW '13, pages 95–106, 2013.

[21] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '13, pages 299–310, 2013.

[22] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. Bosschere. Instruction set limitation in support of software diversity. In P. Lee and J. Cheon, editors, *Information Security and Cryptology ICISC '08*, volume 5461 of *Lecture Notes in Computer Science*, pages 152–165. Springer Berlin Heidelberg, 2009.

[23] R. El-Khalil and A. D. Keromytis. Hydan: Hiding information in program binaries. In *Proceedings of the 6th International Conference on Information and Communications Security*, ICICS '04, pages 187–199, 2004.

[24] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '97, pages 67–72, 1997.

[25] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, 2010.

[26] D. Geer, C. P. Pfleeger, B. Schneier, J. S. Quarterman, P. Metzger, R. Bace, and P. Gutmann. CyberInsecurity: The cost of monopoly – how the dominance of Microsoft's products poses a risk to security. Computer and Communications Industry Association, 2003.

[27] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium*, SEC '12, pages 475–490, 2012.

[28] A. Gupta, S. Kerr, M. Kirkpatrick, and E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. In J. Lopez, X. Huang, and R. Sandhu, editors, *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 293–306. Springer Berlin Heidelberg, 2013.

[29] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 571–585, 2012.

[30] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. librando: Transparent code randomization for just-in-time compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13, pages 993–1004, 2013.

[31] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–11, 2013.

[32] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in Turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies*, WOOT '12, pages 64–76, 2012.

[33] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *Comput. J.*, 23(3):223–229, 1980.

[34] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized NOP insertion. In S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 151–173. Springer New York, 2013.

[35] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 77–98. Springer New York, 2011.

[36] M. Jacob, M. Jakubowski, P. Naldurg, C. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In K. Matsuura and E. Fujisaki, editors, *Advances in Information and Computer Security*, volume 5312 of *Lecture Notes in Computer Science*, pages 100–120. Springer Berlin / Heidelberg, 2008.

[37] 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*, CCS '03, pages 272–280, 2003.

[38] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, 2006.

[39] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques, 2005. http://www.suse.de/~krahmer/no-nx.pdf.

[40] P. Larsen, S. Brunthaler, and M. Franz. Security through diversity: Are we there yet? *IEEE Security & Privacy*, 12(2):28–35, 2014.

[41] C. Linn and S. K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, 2003.

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[43] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, SEC '06, pages 209–224, 2006.

[44] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11(58), 2001. http://www.phrack.org/issues.html?issue=58&id=4.

[45] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007.

[46] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584, 2010.

[47] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 601–615, 2012.

[48] PaX. *Homepage of The PaX Team*, 2001. http://pax.grsecurity.net.

[49] M. Payer. Too much PIE is bad for performance. Technical report, ETH Zürich, 2012.

[50] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM.

[51] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. *Journal of Computer Security*, 18(5):701–749, 2010.

[52] B. Randell. System structure for software fault tolerance. *SIGPLAN Not.*, 10(6):437–449, 1975.

[53] J. Salwan. ROPgadget tool, 2012. http://shell-storm.org/project/ROPgadget/.

[54] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, SEC '11, 2011.

[55] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '03, pages 36–47, 2003.

[56] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[57] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, 2007.

[58] H. Shacham, M. Page, B. Pfaff, E. 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 '04, pages 298–307, 2004.

[59] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 309–318, 2012.

[60] K. Z. Snow, F. Monrose, L. V. 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P '13, pages 574–588, 2013.

[61] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P '13, pages 48–62, 2013.

[62] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[63] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th Interntional Symposium on Recent Advances in Intrusion Detection*, RAID '11, pages 121–141, 2011.

[64] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, 2012.

[65] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28nd Annual Computer Security Applications Conference*, ACSAC '12, pages 299–308, 2012.

[66] T. Wei, T. Wang, L. Duan, and J. Luo. INSeRT: Protect dynamic code generation against spraying. In *Proceedings of the 2011 International Conference on Information Science and Technology*, ICIST '11, pages 323–328, 2011.

[67] D. W. Williams, W. Hu, J. W. Davidson, J. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.

[68] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 79–93, 2009.

[69] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, SEC '13, pages 337–352, 2013.