Verification of Functional Correctness of Code Diversification Techniques*

Jae-Won Jang[1], Freek Verbeek[1,2], and Binoy Ravindran[1]

[1] Virginia Tech, Blacksburg VA, USA
[2] Open University of The Netherlands
{jjang3,freek,binoy}@vt.edu

Abstract. Code diversification techniques are popular code-reuse attacks defense. The majority of code diversification research focuses on analyzing non-functional properties, such as whether the technique improves security. This paper provides a methodology to verify functional equivalence between the original and a diversified binary. We present a formal notion of binary equivalence resilient to diversification. Moreover, an algorithm is presented that checks whether two binaries – one original and one diversified – satisfy that notion of equivalence. The purpose of our work is to allow untrusted diversification techniques in a safety-critical context. We apply the methodology to three state-of-the-art diversification techniques used on the GNU Coreutils package. Overall, results show that our method can prove functional equivalence for 85,315 functions in the analyzed binaries.

Keywords: Code Diversification, Functional Equivalence, Verification

1 Introduction

Generally, a binary is a result of compiling source code into machine code. Binary compilation is a deterministic process where if the source code is unchanged, the resulting binary will remain the same. Due to this deterministic nature, if a binary contains a critical error that adversaries can attack, this issue propagates to all instances of binaries. Code diversification is a software technique used to defend such attacks (e.g., code-reuse [11] or return-oriented programming (ROP) [6, 41]). The technique consists – broadly described – of modifying the compilation process so that the same source code produces different binaries that offer the same functionality. Code diversification ensures that when an attacker has compromised a binary, the knowledge gained cannot easily be reused on the other binaries (of the same type)

There have been a plethora of proposed code diversification techniques presented in the literature [21, 28]. Many of these techniques are based on either recompilation from source-code or rewriting at the machine-code level. Examples include `nop` insertion [20, 24], stack layout randomization [15], and relocation of

* This is the authors version of the work posted here per publisher's guidelines for your personal use. Not for redistribution. The final authenticated version was published in the Proceedings of the 13th International Symposium on NASA Formal Methods, NFM 2021, Virtual, May 24-28, 2021 and is available online at: https://doi.org/10.1007/978-3-030-76384-8_11

Table 1: Covered Code Diversification Techniques

Diversification Technique	**Reference**
Instruction Reordering	[15, 19, 23, 25, 36]
Basic Block Reordering	[9, 15, 27]
Stack Layout Randomization	[2, 15, 17, 31]
`nop` Insertion	[9, 20–22, 24, 28, 46]
Function Reordering	[3, 17, 23, 27]
Static Binary Rewriting	[10, 18, 19, 27, 47]

basic blocks, functions, and instructions [27]. Without exception, these diversification techniques are evaluated on non-functional properties, such as whether the entropy is increased, performance overhead, etc. None of the existing techniques are evaluated on whether the diversification preserves functional equivalence, which reduces trust in deploying such techniques in production settings. Upon diversification, no proof or theorem shows that the diversified binary is functionally equivalent to the vanilla (i.e., original) binary. A key challenge is that strictly speaking, they are *not* functionally equivalent. Various registers and memory locations may contain different values in both worlds. However, the binaries should be *similar* for *relevant* state parts, such as the registers storing in- and output.

In this paper, we propose a definition of functional equivalence between vanilla and diversified binary. The definition is based on stuttering bisimulation [1], thereby showing that the binaries share a large class of temporal properties. Moreover, we propose a methodology that takes an input of a disassembled vanilla and diversified binary and establishes whether that definition holds or provides a counterexample. We fold our methodology in a tool and evaluate with several different code diversification techniques listed in Table 1. The purpose of our work is to allow untrusted diversification techniques to be used in safety-critical context. As a benchmark, we diversify GNU Coreutils with these diversification techniques and show functional equivalence for many functions. The main limitation is that we are not able to deal with indirect branching. Moreover, we assume the a binaries can be successfully disassembled.

The state-of-the-art provides various techniques to compare two binaries [7, 12, 33, 35, 39, 42, 50]. A broad class of the comparison techniques is statistical or learning-based [13, 48, 50]. They give a *probability* instead of a *guarantee* that the two binaries are related. Luo et al. [35] provide a binary similarity comparison technique that is resilient to code diversification. Another class of comparison techniques focuses on strong equivalence (e.g., requiring that the same registers are used for the same variables) [7, 39]. They thus cannot deal with code diversification since this produces two binaries that are not strongly equivalent. In Section 6, we provide a further discussion of the state-of-the-art. A verified diversifier is out of reach of the current state-of-the-art: a diversifier typically is based on `gcc` or `clang` with specific compiler options. Thus verifying the diversifier implies verifying one of those compilers. Moreover, such an approach would limit the applicability of the methodology to that specific diversifier. For these reasons, we consider the *output* of some *black-box* and *untrusted*

```
1  | void *foo();
2  | void *bar();

3  | int main(int argc,
      char** argv)
4  | {
5  |  int a = 0;
6  |  if (a == 0)
7  |     {
8  |         foo();
9  |     }
10 |  else
11 |     {
12 |         bar();
13 |     }
14 |  return 0;
15 | }

16 | void *foo()
17 | {
18 |  printf("Foo\n");
19 | }

20 | void *bar()
21 | {
22 |  int c = 17;
23 | }
```

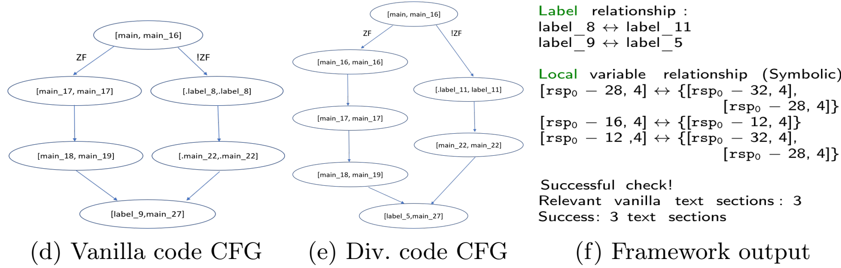
(a) Source code

```
1  | main:
2  |   push rbp
3  |   mov  rbp, rsp
4  |   sub  rsp, 0x30
5  |   mov  dword ptr [rbp-0x4], 0
6  |   mov  dword ptr [rbp-0x8], edi
7  |   jne  .label_8
8  |   call foo
9  |   mov  qword ptr [rbp-0x20], rax
10 |   jmp  .label_9
11 | .label_8:
12 |   call bar
13 |   mov  qword ptr [rbp-0x28], rax
14 | .label_9:
15 |   xor  eax, eax
        ...
19 | foo:
        ...
24 |   mov  dword ptr [rbp-0xc], eax
25 | bar:
26 |   push rbp
27 |   mov  rbp, rsp
28 |   mov  dword ptr [rbp-0xc], 0x11
29 |   mov  rax, qword ptr [rbp-0x8]
```

(b) Disassembled Code

```
19 | foo:
       ...
24 |   mov dword ptr [rbp-0x4], eax
       ...
11 | .label_11:
      nop
12 |   call  bar
13 |   mov qword ptr [rbp-0x28], rax
14 | .label_5:
15 |   xor eax, eax
       ...
1  | main:
2  |   push  rbp
3  |   mov rbp, rsp
      mov rbp, rbp
4  |   sub rsp, 0x30
5  |   mov dword ptr [rbp-0x14], 0
5  |   mov dword ptr [rbp-0x04], edi
7  |   jne .label_11
      lea rdi, [rdi]
8  |   call  foo
9  |   mov qword ptr [rbp-0x20], rax
10 |   jmp .label_5
25 | bar:
26 |   push  rbp
27 |   mov rbp, rsp
28 |   mov dword ptr [rbp-4], 0x11
      mov rbp, rbp
29 |   mov rax, qword ptr [rbp-0x10]
```

(c) Disassembled div. code



(d) Vanilla code CFG (e) Div. code CFG

```
Label  relationship :
label_8 ↔ label_11
label_9 ↔ label_5

Local  variable  relationship  (Symbolic):
[rsp0 − 28, 4] ↔ {[rsp0 − 32, 4],
                  [rsp0 − 28, 4]}
[rsp0 − 16, 4] ↔ {[rsp0 − 12, 4]}
[rsp0 − 12 ,4] ↔ {[rsp0 − 32, 4],
                  [rsp0 − 28, 4]}

Successful  check!
Relevant  vanilla  text  sections :  3
Success:  3  text  sections
```

(f) Framework output

Fig. 1: Example

diversifiers and verify whether the diversified binary is functionally similar to its vanilla version.

The main contributions of our work are:

1. A formal definition of functional equivalence resilient to code diversification
2. A scalable methodology for establishing that equivalence between a vanilla and a diversified binary
3. The application of this methodology to different diversification techniques on GNU Coreutils binaries

To the best of our knowledge, this paper is the first to present a formal definition of functional equivalence for diversified binaries and to verify that equivalence.

This paper is organized in the following structure. Section 2 presents an overview of our method. In Section 3, we formally define the equivalence relation, followed by the algorithm for equivalence checking in Section 4. The evaluation and limitations of our work are presented in Section 5. Section 6 presents related work and compares them with our work. Lastly, we conclude in Section 7.

2 System Overview

This section provides an overview of the input, intermediate steps and the output of our method, using Figure. 1 as a running example. Note that the C code in

Figure. 1a is shown for the sake of presentation only: it is not required as input to our method; we use this example to demonstrate how a diversification technique affects the original binary.

2.1 Disassembly

The methodology takes as input a vanilla and a diversified (div.) binary. The first step for our methodology is disassembly (see Figures 1b and 1c). An example of diversification is shown in Figure. 1c, where text sections (e.g., `main, foo, bar`) are reordered due to function reordering. A key feature of disassembly that we require is *symbolization*. Symbolization replaces concrete addresses with symbolic values, i.e., labels. Binaries are usually compiled (or translated into machine-readable code) from high-level source code. After compilation, it goes through the assembler to generate an object file (architecture-dependent), which is linked with different libraries to create a binary executable. During this process, a large portion of useful high-level information disappears. Specifically, relocation information disappears, which a linker uses to maintain the coherence between multiple libraries that the binary calls. Symbolization aims at recovering relocation-related information.

An example of symbolization is shown in Lines 11 and 14 of Figures 1b and 1c. The text section names of `main, foo` and `bar` are preserved by compilation from the source code. The labels (`.label_8` and `.label_9`), however, are not present in the original binary. They are the result of symbolizing concrete addresses.

We use `Ramblr` for disassembly and symbolization [45]. `Ramblr` is a subset of an open-source disassembler framework `angr`. `angr` reliably disassembles the binary, and this was demonstrated on a set of binaries from the DARPA cyber grand challenge (a computer security competition dealing with malicious binaries) [43].

2.2 Control-Flow Graph Construction

The second step is the CFG construction. We extract CFGs from the binary using an off-the-shelf algorithm similar to `angr`'s CFGFast [43]. A CFG consists of nodes and edges. A node consists of a basic block, (i.e., a list of instructions). Edges are labeled with flags. We construct one CFG per text section (for both vanilla and diversified binaries). The reason for this is because our methodology proves functional equivalence per text section rather than iterating through the complete CFG of an entire binary.

The vanilla and diversified binaries are supposed to be functionally equivalent, but as Figures 1d and 1e show, the diversified CFG is different. The process of `nop` insertion [20] actually inserted a `lea` instruction just before the `call foo` statement (between Lines 7–8 in Figure. 1c). This causes the difference in CFGs, and it complicates the functional equivalence verification process, as we cannot only check whether the two CFGs are equal. Instead, we must check whether they are stuttering bisimilar.

2.3 Local Variable Normalization

Variables are storage locations that the program can manipulate. Local variables are referenced by an offset from special-purpose registers (the frame pointer `rbp`

or the stack pointer `rsp`). Register `rbp` points to the top of the current stack frame, and `rsp` points to the bottom.

After constructing CFGs, we normalize local variables to prepare for our comparison algorithm. Normalizing in this context means that local variable offsets become relative to the *initial* value of the stack ($\texttt{rsp}_0$) pointer. For example, in Figure. 1b, the memory address $\texttt{rbp} - \texttt{0x4}$ is accessed at Line 5. Since the frame pointer is a copy of the stack pointer (Line 3), which has been decremented with 8, this memory address becomes $\texttt{rsp}_0 - \texttt{0x12}$ after normalization. Normalization allows us to compare memory locations that are shuffled.

2.4 CFG Comparison

Bisimulation is a relation between two transition systems, where one can simulate the others' (transitions) and vice versa. In particular, we use *divergence-sensitive stuttering bisimulation* [1,4] for our work. To check whether two CFGs are stuttering bisimilar, we treat them as transition systems and check whether they are stuttering bisimulation-equivalent. In other words, we check if there exists a stuttering bisimulation between both transition systems.

The intuition behind a stuttering bisimulation is that it deals with internal steps, steps that perform no visible behavior. Stuttering bisimulation allows two transition systems to be equivalent regardless of how many internal steps it takes to get to the next state, and mainly deals with basic blocks consisting solely of instructions that perform no state change, such as a basic block [`main_18, main_19`] in Figure. 1e. Divergence-sensitivity prevents a situation where one of the transition systems permanently executes internal steps, whereas the other does not.

A divergence-sensitive stuttering bisimulation is a strong notion of equivalence, as it preserves a large set of properties. This set of properties includes safety, liveness, and reachability properties. Formally, the set of properties preserved is computation tree logic (CTL) except the next operator, i.e., $\text{CTL}^* \backslash \mathbf{X}$ [1]. In words, this means that any CTL^* property is preserved as long as it does not concern specifically the current branching decision.

2.5 Output

If our methodology finds no counterexample, it provides an output stating that the binaries are functionally equivalent (see Figure. 1f). In addition, our work offers evidence for that claim, which is a mapping between vanilla and diversified parts of the binaries. If we find any potential issues (e.g., limitations or problems), our work can provide the output stating the basic block causing the discrepancy. A third possible output is "Unsupported" (see Section 5.2).

When a program becomes diversified by a technique such as reordering of instruction or basic blocks, relocation information of such a program gets affected. For example, `.label_8` and `.label_11` from Figures 1b and 1c are semantically equivalent (Line 11 for both), but different in label names. The CFG comparison algorithm thus keeps track of a map relating labels in the vanilla world to labels in the diversified world. Similarly, stack shuffling may cause local variables to be put into different memory regions. A mapping is maintained that relates

diversified memory regions to vanilla ones. For example, the 4-byte vanilla memory region $[\mathtt{rsp}_0 - 16, 4]$ (accessed at Line 6 in Figure. 1b) is mapped to the diversified memory region $[\mathtt{rsp}_0 - 12, 4]$ (accessed at Line 6 in Figure. 1c).

3 Soundness of Code Diversification

This section defines an equivalence relation over binaries. We will call a code diversification effort *sound* if the vanilla binary and the produced diversified binary are equivalent under that relation.

The key idea is to establish a *divergence-sensitive stuttering bisimulation* over the transition systems modeling the two binaries [1]. Formally, a transition system TS is defined by a tuple $\langle S, \leadsto, I\rangle$. Here S is a set of states, $\leadsto$ of type $S \times S \mapsto \mathbb{B}$ is a transition relation, and I is a set of initial states.

Note that this definition omits either a labeling function or actions on edges. Typically, definitions of stuttering bisimulation are based on one of these two. However, due to stack-frame shuffling, neither of these is convenient. For example, let both the vanilla and diversified binaries have two labeling functions L_0 and L_1. Both translate concrete states to atomic propositions. Stuttering bisimulation then considers two states s and s' to be equal only if $L_0(s) = L_1(s')$. Consider again function `main` in Figures 1b and 1c. The labeling functions should translate states to atomic propositions in such a way that the value stored at the vanilla memory region $[\mathtt{rsp_0} - \mathtt{16}, \mathtt{4}]$ and the value stored at the diversified memory region $[\mathtt{rsp_0} - \mathtt{12}, \mathtt{4}]$ map to the same atomic proposition. The labeling function is thus hard to define, as it is dependent on the relation established between the two binaries. Instead, we will formalize a *state comparison function* $\doteq$ of type $S_0 \times S_1 \mapsto \mathbb{B}$, and define stuttering bisimilarity relative to the given state comparison function.

Definition 1. *Let TS_0 and TS_1 be two transition systems, and let $\doteq$ of type $S_0 \times S_1 \mapsto \mathbb{B}$ be a state comparison function. Binary relation $\mathcal{B}$ of type $S_0 \times S_1 \mapsto \mathbb{B}$ is a* divergence-sensitive stuttering bisimulation wrt. $\doteq$, *if and only if, for any states $s_0 \in S_0$ and $s_1 \in S_1$, such that $\mathcal{B}(s_0, s_1)$:*

1. $s_0 \doteq s_1$
2. *if $s_0 \leadsto_0 s_0'$ and $\neg\mathcal{B}(s_0', s_1)$, then there exists a finite path fragment $[s_1, t_0 \ldots t_n, s_1']$ such that $\mathcal{B}(s_0, t_i)$ for all i and $\mathcal{B}(s_0', s_1')$*
3. *if $s_1 \leadsto_1 s_1'$ and $\neg\mathcal{B}(s_0, s_1')$, then there exists a finite path fragment $[s_0, t_0 \ldots t_n, s_0']$ such that $\mathcal{B}(t_i, s_1)$ for all i and $\mathcal{B}(s_0', s_1')$*
4. *there exists an infinite path fragment $[s_0, t_0, t_1 \ldots]$ such that $\mathcal{B}(t_i, s_1)$ for all i, if and only if, there exists an infinite path fragment $[s_1, u_0, u_1 \ldots]$ such that $\mathcal{B}(s_0, u_j)$ for all j.*

Two transition systems are divergence-sensitive stuttering bisimilar wrt. $\doteq$, notation $TS_0 \approx TS_1$, if and only if there exists a divergence-sensitive stuttering bisimulation $\mathcal{B}$ that relates all initial states, i.e., for all $s_0 \in I_0$, there exists some $s_1 \in I_1$ such that $R(s_0, s_1)$, and the other way around.

Soundness of diversification is expressed as a property over CFGs. We assume the existence of a function cfg that takes as input a binary and produces a CFG.

Formally, a CFG consists of a tuple $\langle B, \rightharpoonup, e\rangle$, where B denotes a set of basic blocks, $\rightharpoonup$ of type $B \times B \mapsto \mathbb{B}$ denotes a transition relation, and e of type B denotes the entry point. The start address of a basic block b can be accessed via $b.\texttt{addr}$, the list of instructions via $b.\texttt{instrs}$.

We consider the transition system corresponding to a given CFG. The transition system consists of concrete states that map registers, 64-bit byte-addressable memory, and flags to values. We use S_C to denote the set of concrete states, R to denote the set of registers, and A to denote the address space. Given a concrete state s, we use $s.\texttt{rip}$ to denote the value stored in register `rip`, and similar for other registers and flags. Notation $s.\texttt{mem}(a)$ returns given a 64-bit address a the byte-value stored in the memory at that address. Function run of type $B \times S_C \mapsto \{S_C\}$ takes as input a basic block and the current concrete state, and runs the list of instructions in the basic block. It returns the set of possible next concrete states. Since a basic block does not have loops, this function terminates.

Definition 2. *Let $g = \langle B, \rightharpoonup, e\rangle$ be a CFG. The transition system of g, notation $\overline{g}$, is defined by $\langle S_C, \rightsquigarrow, I\rangle$. Here set of initial concrete states I is defined as follows:*

$$I \stackrel{def}{=} \{s \mid s.\texttt{rip} = e.\texttt{addr}\}$$

and transition relation $\rightsquigarrow$ is constructed as follows:

$$\frac{s' \in \mathit{run}(b, s) \wedge b \rightharpoonup b' \wedge s.\texttt{rip} = b.\texttt{addr} \wedge s'.\texttt{rip} = b'.\texttt{addr}}{s \rightsquigarrow s'}$$

In words, the transition system $\overline{g}$ starts at states whose instruction pointer `rip` is equal to that of the first instruction of the basic block that is the CFGs' entry point. It then moves from state to state by executing entire basic blocks. The transition system is thus at the same granularity as the CFG.

Definition 1 depends on a state comparison function. The stronger this comparison function, the stronger the equivalence. As illustrated, if $s_0 \doteq s_1$ is true for any state, then any two transition systems are bisimilar. We thus define a comparison function for concrete states that is as strong as possible. Ideally, we want to compare all state parts, i.e., all registers, memory, and flags. In practice, we consider only the set of *relevant* registers. For instance, the instruction pointer (`rip`) is irrelevant: in the two worlds, it will differ since the executed instructions have different addresses due to, e.g., `nop` insertion. The frame-and stack-pointers are irrelevant since stack frame shuffling can enlarge the stack frame. Which registers are relevant may depend on the current state. This is modeled with a function $\mathcal{R}$ that returns relevant registers (e.g., all registers except the irrelevant ones) given the current state. In practice, we ignore flags: their impact on execution is covered by proving that the same branching decisions are made. Finally, we need to map diversified memory addresses to their vanilla counterparts. This is modeled with a mapping $\mathcal{M}$.

Definition 3. *Let $\mathcal{R}$ of type $S_C \mapsto \{R\}$ be a function that returns a set of relevant registers given a concrete state. Let $\mathcal{M}$ of type $A \mapsto A$ be a mapping from diversified memory addresses to vanilla memory addresses. The* concrete state comparison

```
1  | main:
2  |  push rbp
3  |  mov  rbp, rsp
4  |  sub  rsp, 0x30
5  |  lea rdi, [rdi]
6  |  xor eax, eax
7  |  mov dword ptr [rbp - 0x14], eax
8  |  mov  dword ptr [rbp - 0x4], edi
9  |  cmp  dword ptr [rbp - 0x4], esi
10 |  jne  .label_22
```

(a) Diversified `main` Basic Block

$$
\begin{aligned}
&\texttt{rsp} &&:= \texttt{rsp} - 0x38\\
&\texttt{rbp} &&:= \texttt{rsp} - 0x8\\
&\texttt{rax} &&:= 0\\
&[\texttt{rsp} - \texttt{0x8}, 8] &&:= \texttt{rbp}\\
&[\texttt{rsp} - \texttt{0x1c}, 4] &&:= 0\\
&[\texttt{rsp} - \texttt{0xc}, 4] &&:= \langle 31, 0\rangle(\texttt{rdi})\\
&\texttt{ZF} &&:= \langle 31, 0\rangle(\texttt{rdi}) = \langle 31, 0\rangle(\texttt{rsi})\\
&\texttt{CF} &&:= \langle 31, 0\rangle(\texttt{rdi}) < \langle 31, 0\rangle(\texttt{rsi})\\
&\texttt{SF} &&:= \langle 31, 0\rangle(\texttt{rdi}) <_s \langle 31, 0\rangle(\texttt{rsi})
\end{aligned}
$$

(b) Symbolic Execution Output

Fig. 2: Symbolic Execution Example

function *of* M*, notation* $\stackrel{c}{=}_{\langle\mathcal{R},\mathcal{M}\rangle}$*, is defined as follows:*

$$s_0 \stackrel{c}{=}_{\langle\mathcal{R},\mathcal{M}\rangle} s_1 \stackrel{def}{=} \begin{cases} \quad \forall \mathbf{r} \in \mathcal{R}(s_0) \cdot s_0.\mathbf{r} = s_1.\mathbf{r} \\ \wedge \ \forall a \in A \cdot s_0.\mathtt{mem}(a) = s_1.\mathtt{mem}(\mathcal{M}(a)) \end{cases}$$

In words, two concrete states are considered equal if all relevant registers are equal and all memory in both worlds is the same after mapping diversified addresses to vanilla addresses. For example, in Figure. 1, the values stored at vanilla address $\mathtt{rsp}_0 - \mathtt{16}$ and diversified address $\mathtt{rsp}_0 - \mathtt{12}$ will be compared.

Definition 4. *Let B be a binary and let D be a diversification function that takes as input a binary and produces a diversified binary. Diversification D is sound for binary B, if and only if, there exists a function $\mathcal{R}$ and mapping $\mathcal{M}$ such that the transition systems of the CFGs are divergence-sensitive stuttering bisimilar wrt. the concrete state comparison function of $\mathcal{R}$ and $\mathcal{M}$.*

$$\mathrm{sound}(D, B) \stackrel{def}{=} \exists \mathcal{R}, \mathcal{M} \cdot \overline{\mathrm{cfg}(B)} \approx \overline{\mathrm{cfg}(D(B))} \ \textit{wrt.} \ \stackrel{c}{=}_{\langle\mathcal{R},\mathcal{M}\rangle}$$

4 Algorithm

In order to check soundness (see Definition 4), a witness must be found for function $\mathcal{R}$ and mapping $\mathcal{M}$. This section presents an algorithm to find these witnesses. It consists of four steps:

1. Produce the function $\mathcal{R}$ and mapping $\mathcal{M}$ by running symbolic execution on each basic block in the two CFGs; for each symbolic block, keep track of the relevant registers.
2. Express all local variables in terms of the initial value of the stack pointer $\mathtt{rsp}_0$.
3. Check for stuttering bisimulation on the symbolized CFGs, while keeping track of memory mapping $\mathcal{M}$.

Step 1 (Symbolic Execution) The purpose of symbolic execution is to express the semantics of each basic block in a way that is independent of the actual instructions. Figure. 2 depicts an example of symbolic execution where we show a basic block and its symbolic output. The symbolic output consists of assignments of symbolic expressions to state parts (registers, memory flags). Symbolic

expressions consist of, among others, immediate values, reading from state parts, and common bit-vector operations. These operations include taking bit subsets, concatenation, logical operators, casting operators, floating-point operators, and signed and unsigned arithmetic. For example, after the execution of the basic block in Figure. 2, register `rax` (`eax` is the lower 32 bits of `rax`) has become 0, and the sign flag is set by a signed integer comparison of the lower 32 bits of the `rdi` and `rsi` registers. All values are relative to the initial state of the basic block. For example, the instruction at Line 8 uses the frame pointer `rbp`, but since that value at that line is equal to the initial stack pointer minus 8, the instruction results in a write to the symbolic memory region $[\texttt{rsp} - \texttt{0xc}, 4]$.

In Figure. 2b, it can be seen that symbolic execution produces a result largely agnostic of diversification. For the sake of presentation, the basic block has been manually modified with two features found in typical diversification tools. At Line 5 in Figure. 2a, a `lea` instruction has been inserted that performs no state change: it is effectively a `nop`. Instead of directly writing 0 to memory with one instruction, we use two instructions (Lines 6 and 7). First, `xor` is used to write zero to register `eax` (which denotes the lower 32 bits of register `rax`), and then `mov` is used to do the memory write. Symbolization does not reflect these modifications since they do not influence the semantics of the basic block. The basic block without the manual modifications would have produced the same symbolic output. Finally, the example shows that to compare a vanilla and a diversified basic block, only the register modified in the vanilla world is to be considered relevant. Register `rax` would not have been part of the symbolic output for the non-modified basic block. However, all state parts that are modified in the vanilla world are similarly modified in the diversified world.

Formally, a symbolic state σ consists of registers, memory and flags (we use s and σ for resp. concrete and symbolic states). For each modified register $\texttt{r}$, notation $\sigma.\texttt{r}$ returns a symbolic expression. The set of modified registers is returned by $\sigma.\texttt{regs}$. The memory is modeled by assigning symbolic expressions to symbolic memory regions. Consider again example in Figure. 2b. The notation $\sigma.[\texttt{rsp} - \texttt{0x8}, 8]$ returns the symbolic expression `rbp`. The set of modified memory regions is denoted by $\sigma.\texttt{mems}$.

The algorithm will compare symbolic states instead of concrete ones. This requires us to formulate a *symbolic* state comparison function, which is the symbolic counterpart to its concrete version (see Definition 3).

Definition 5. *Let $\mathcal{N}$ be a mapping from diversified symbolic memory regions to vanilla symbolic memory regions. The* symbolic state comparison function *of $\mathcal{N}$, notation $\stackrel{s}{=}_{\mathcal{N}}$, is defined as follows:*

$$\sigma_0 \stackrel{s}{=}_{\mathcal{N}} \sigma_1 \stackrel{def}{=} \begin{cases} \forall \texttt{r} \in \sigma_0.\texttt{regs} \cdot \sigma_0.\texttt{r} = \sigma_1.\texttt{r} \\ \wedge\ \forall \texttt{r} \in \sigma_0.\texttt{mems} \cdot \sigma_0.\texttt{r} = \sigma_1.\mathcal{N}(\texttt{r}) \end{cases}$$

Mapping $\mathcal{N}$ is defined over symbolic memory regions and symbolic expressions. Given a concrete state, the memory regions, all symbolic values can be concretized. Function γ takes as input a symbolic mapping $\mathcal{N}$, and produces a concrete mapping $\mathcal{M} = \gamma(\mathcal{N})$. For example, if diversified memory region

$[\mathtt{rsp} - \mathtt{8}, \mathtt{8}]$ is mapped by $\mathcal{N}$ to vanilla region $[\mathtt{rsp} - \mathtt{16}, \mathtt{8}]$, then $\mathcal{M}$ will relate 8 individual concrete addresses based on the values of the concrete stack pointers.

Definition 6. *Symbolic execution is sound, if and only if, for any basic blocks* b_0 *and* b_1*, assumption*

$$\mathrm{se}(b_0) \stackrel{S}{=}_{\mathcal{N}} \mathrm{se}(b_1)$$

implies:

$$\forall s_0, s_1 \in S_C \;\cdot\; s_0.\mathtt{rip} = b_0.\mathtt{addr} \;\wedge\; s_1.\mathtt{rip} = b_1.\mathtt{addr} \Longrightarrow \mathit{run}(b_0, s_0) \stackrel{C}{=}_{\langle \mathcal{R}, \mathcal{M} \rangle} \mathit{run}(b_1, s_1)$$

where

$$\mathcal{R}(s_0) = \mathrm{se}(b_0).\mathtt{regs}$$
$$\mathcal{M} \quad = \gamma(\mathcal{N})$$

In words, comparing two symbolic states should suffice to show successful comparison of all the concrete states they represent. The set of relevant registers $\mathcal{R}$ is the set of all modified registers in the vanilla world. The concrete memory map $\mathcal{M}$ is obtained by concretizing the symbolic memory region map.

Step 2 (substitute for $\mathtt{rsp_0}$) Local variables are addressed relative to either the stack pointer `rsp` or the frame pointer `rbp`. The values in these registers are not static, i.e., they change during the execution of the function. This change complicates formulating a static address mapping $\mathcal{M}$. Consider, Lines 5 and 9 of Figure. 1b, which deal with addresses $[\mathtt{rbp} - \mathtt{0x4}]$ and $[\mathtt{rbp} - \mathtt{0x20}]$. In between these lines, the value of register `rbp` may have changed in such a way that these addresses are actually equal. To formulate a static mapping $\mathcal{M}$, we thus make all local variables relative to the initial value of the stack pointer $\mathtt{rsp}_0$. This is achieved by propagating two substitutions through the symbolized CFG. Initially, these two substitutions are $\mathtt{rsp} \coloneqq \mathtt{rsp}_0$ and $\mathtt{rbp} \coloneqq \mathtt{rbp}_0$. Thus, for the entry block, any occurrence of the stack pointer is simply replaced by $\mathtt{rsp}_0$. The substitutions are updated if the current basic block updates either the stack- or frame pointer. In the example, after the first basic block, the current substitution is $\mathtt{rsp} \coloneqq \mathtt{rsp}_0 - \mathtt{0x38}$ and $\mathtt{rbp} \coloneqq \mathtt{rsp}_0 - \mathtt{0x8}$.

In this fashion, substitutions are propagated through the CFG. In case of an encountering a visited node (i.e., a loop or paths in the CFG converging), it is verified that the current substitution is equal to the substitution already applied to the visited node. If this holds any time a visited node is encountered, then the current substitutions constitute an invariant.

Step 3 (checking for stuttering bisimulation) Algorithm 1 presents a procedure Check that compares two CFGs. It takes as input the current basic blocks – initially starting with the entry points – and traverses the CFGs simultaneously. It provides as output a Boolean indicating the existence of a stuttering bisimulation (Line 19). Before we explain the algorithm in more detail, we introduce some definitions.

Definition 7. *Let* b_0 *and* b_1*, be two basic blocks (vanilla and diversified respectively). Branching for basic blocks is equivalent, if and only if:*

$$\mathsf{eq_branching}(\mathsf{b_0}, \mathsf{b_1}) \stackrel{def}{=} \forall f \cdot (\exists b_0' \cdot b_0 \stackrel{f}{_} b_0') \Leftrightarrow (\exists b_1' \cdot b_1 \stackrel{f}{_} b_1')$$

Algorithm 1 Check Between Vanilla and Diversified CFGs

```
1: function Check(b0, b1)
2:     if b0 and b1 are unvisited then
3:         mark b0 and b1 as visited
4:         σ0 = se(b0)
5:         σ1 = se(b1)
6:         Update (N, σ0, σ1)
7:         if σ0 ≐_N σ1 ∧ eq_branching(b0, b1) then
8:             for each (b'0, b'1) ∈ get_children(b0, b1) do
9:                 Check (b'0, b'1)
10:            end for
11:        else if is_skip(b1) then
12:            mark b1 as visited
13:            Check (b0, b'1) with b1 ⇀ b'1
14:        else
15:            mark current b1 text section as counterexample
16:            return False
17:        end if
18:    else
19:        return True
20:    end if
21: end function
```

In words, equal branching returns true if both blocks have the same number of children with the same flags.

Definition 8. *Let b_0 and b_1, be two basic blocks (vanilla and diversified respectively). The set of children of b_0 and b_1 is defined as follows:*

$$\mathsf{get_children}(\mathsf{b_0}, \mathsf{b_1}) \overset{def}{=} \{(b'_0, b'_1) \cdot \exists f \cdot b_0 \overset{f}{_} b'_0 \wedge b_1 \overset{f}{_} b'_1\}$$

In words, *get_children* returns the set of children that is to be explored from current basic blocks b_0 and b_1.

Definition 9. *Basic block b is a* skip *if and only if:*

$$\mathsf{is_skip}(b) \overset{def}{=} \sigma.\mathtt{regs} \subseteq \{\mathtt{rip}\} \wedge \sigma.\mathtt{mems} = \emptyset$$

In words, a basic block is a skip if it does not modify memory and the only register that is modified (if any) is the instruction pointer `rip`.

Algorithm 1 essentially is a simultaneous depth-first exploration. If both basic blocks are flagged as visited, a stuttering bisimulation for the current basic blocks b_0 and b_1 has been established. If not, both basic blocks are flagged as visited, and the comparison continues. Each block is first symbolically executed, producing symbolic states σ_0 and σ_1. The currently established stuttering bisimulation relation is updated (Line 6). This update stores that from now on, $\mathcal{R}(s_0)$ returns the set of registers modified by basic block b_0, i.e., $\mathcal{R}(s_0) = \sigma_0.\mathtt{regs}$ for all states s_0 such that $s_0.\mathtt{rip} = \sigma_0.\mathtt{rip}$. Moreover, memory mapping $\mathcal{N}$ is updated.

If the two basic blocks are semantically equivalent and they have equal branching (Line 7) the check proceeds by exploring all children. If not, then the current diversified basic block may be a skip. In that case, the check proceeds by comparing the current vanilla basic block b_0 to the child of the skip b'_1. If diversified basic block b_1 was not a skip, then a discrepancy has been found and the algorithm returns false (Line 16).

Theorem 1. *Let $g_0 = \langle B_0, __{}_0, e_0 \rangle$ and $g_1 = \langle B_1, __{}_1, e_1 \rangle$ be the control flow graphs of the vanilla and diversified binaries respectively. Let $\mathcal{R}$ and $\mathcal{M} = \gamma(\mathcal{N})$ be the mappings produced by the algorithm. The algorithm decides a divergence-blind stuttering bisimulation:*

$$\textsc{Check}(e_0, e_1) \longleftrightarrow g_0 \approx g_1 \ \textit{wrt.} \ \stackrel{c}{=}_{\langle \mathcal{R}, \mathcal{M} \rangle}$$

Proof. Soundness of the algorithm is based on the work of Fernandez et al. [14]. In that paper, it is shown that a bisimulation can be decided by a depth-first search that explores two transition systems simultaneously. Proposition 3.2 of that paper states that two deterministic transition systems are bisimilar, if and only if, a simultaneous depth-first search is not able to find a path to a pair of states with a different number of children. That is exactly what is verified by our algorithm. A key difference is that the work of Fernandez et al. is formulated for an algorithm checking strong bisimulation. Line 11 of Algorithm 1 adds an additional case for dealing with stuttering steps: the diversified world can do an arbitrary number of skips before a bisimilar node is encountered. Divergence-sensitivity is guaranteed by checking whether the number of children is always equivalent for all encountered bisimilar nodes (Line 7).

5 Evaluation

The methodology is applied to all 93 binaries of `GNU coreutils 8.31`. The source code is compiled on Linux Ubuntu 16.04 x86-64 using the `clang` with a variety of optimization levels (`-O0` to `-O3`) (depending on the diversification tool). All experiments are run on a machine with an AMD FX-8350 CPU and 8 GB RAM. `GNU Coreutils` is a suitable benchmark as it contains realistic and sufficiently complex binaries, ranging from small (10,692 assembly LOCs) to large (133,065 assembly LOCs).

We cover three off-the-shelf diversification tools which, combined, cover all techniques listed in Table 1: 1) `nop` insertion [20], 2) Compiler-assisted Code Randomization (CCR) [27], and 3) stack shuffling [15]. `nop` insertion inserts `nop` in front of targeted instruction [20]. CCR [27] leverages a compiler-rewriter process to transform inserted metadata into security primitive. The stack shuffling technique [15] diversifies the stack layout per binary.

5.1 Results

Table 2 presents the results. Consider the data for the `CCR` tool on binaries compiled with `-O1`. Out of 93 binaries, 87 binaries were analyzed. The remainder is split into one binary that could not be diversified by `CCR` (column Not Div.) and five binaries that could not be disassembled by `Ramblr` (column Not Dis.). Of the 11769 text sections of the 87 analyzed binaries, 10808 text sections were analyzed and proven to be soundly diversified. Zero text sections were shown to be unsoundly diversified, and 961 text sections contained behavior unsupported by our tool (see Section 5.2).

The table shows we did not find a diversification issue in any of the binaries. However, for each of the listed techniques in Table 1, we manually introduced

Table 2: The evaluation of our methodology on GNU Coreutils v8.31

Diversification Tools		Analyzed	Not Div.	Not Dis.	Sound	Counterexample	Unsupported
		Binaries			Text Sections		
`nop` ins. [20]	`-O0`	93	0	0	12966	0	546
	`-O1`	91	0	2	7427	0	831
	`-O2`	88	0	5	7519	0	930
	`-O3`	88	0	5	7694	0	1079
CCR [27]	`-O0`	93	0	0	12805	0	697
	`-O1`	87	1	5	10808	0	961
	`-O2`	83	2	8	6681	0	868
	`-O3`	79	8	6	6619	0	836
S. shuf. [15]	`-O0`	93	0	0	12796	0	706

Binaries = result with respect to the number of binaries

Text Sections = result with respect to the number of text sections from analyzed binaries

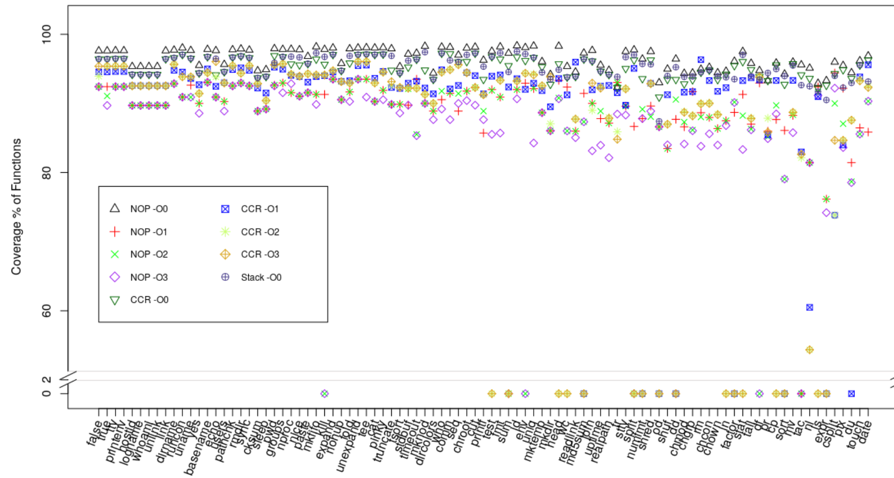


Fig. 3: The coverage rate per diversification techniques over all GNU `Coreutils`

bugs indicative of that technique. For example, we manually inserted one `nop` too many in the diversified version of the `wc` program and reran our tool. For each inserted bug, our methodology reports a counterexample. This report provides information on which text section that the bug is in. Moreover, it also gives the line number of parsed disassembled code for both vanilla and diversified binaries for in-depth debugging purposes.

Figure. 3 shows, per binary, the percentage of text sections proven to be soundly diversified. The binaries are sorted from the least number of assembly LOCs (`false`, 10,692 LOCs) to the largest number of LOCs (`date`, 133,065 LOCs). The zero-outliers are the 31 cases where a binary could either not be diversified or be disassembled for the given diversification technique and the given optimization level. For the remaining binaries the average coverage is 91.26%

with a minimum of 87.7% for `nop` insertion with optimization `O3` and a maximum of 95.95% for `nop` insertion without optimizations.

5.2 Limitations

Our work's main limitation is that we cannot deal with indirect branching (e.g., indirect jumps). Indirect branching occurs when jump- or call-addresses are computed dynamically, and solving indirect branching requires assembly-level invariants on the values involved in that computation. Second, we cannot reason about shuffled local arrays, e.g., due to an `alloca` statement,. The key problem here is that the array-size is not known at the assembly-level. Third, as the optimization level increases, a compiler performs expensive analyses and applies more aggressive transformations (e.g., perform a scalar replacement or loop transformations) to improve the binary. Therefore, as Figure. 3 shows, as the optimization level increases, proving functional equivalence becomes harder, due to optimized instructions (e.g., packed instructions such as `puncpkhbw`). Lastly, if a disassembler is unable to disassemble the binary, then our tool cannot do any verification. Currently, our methodology only supports *diversification* techniques. Proving the functional equivalence between *code obfuscated* binaries to the vanilla binaries is more difficult due to the convoluted modifications that obfuscation techniques do to camouflage the code.

6 Past and Related Work

Our work is closely related to the topic of software similarity detection, which has been widely studied in the context of code plagiarism, clone detection, bug finding, and identifying zero-day vulnerability.

Static and dynamic analysis. There are static and dynamic approaches to analyze software similarity. Static approach analyzes the code without executing it [40], while dynamic approach run functions of the target binaries with the same input and dynamically measure similarity [13]. Among many static approaches, our work is closely related to the graph-based method. [5, 26, 32, 34, 48]. Graph-based methods parse code into CFGs whose subtrees are searched using different graph matching techniques to obtain matching pairs. Lim and Nagarakatte [32] checked for equivalence using a graph-based methodology and symbolic execution. However, this work focused specifically on cryptographic algorithms and does not deal with diversified binaries.

Equivalence checking using symbolic execution. Various research projects use symbolic execution to prove equivalence, such as `BinHunt` [16], `KLEE` [39], and many others [33, 38, 44]. `KLEE` checks, among others, code equivalence. However, this work is source-code based. The most similar work to ours is `BinHunt`, where the authors propose a technique based on backtracking to find semantical differences in binaries that have different register allocation and basic block re-ordering. However, they do not deal with instruction reordering and stack-frame shuffling. Moreover, they do not provide a formal definition of their soundness criterion and therefore do not show what class of properties is preserved between the vanilla and the diversified binaries. The other papers are similar in that their

focus lies on different subjects such as finding bugs or equivalence checking between cross-architecture. Lastly, `CoP` [35] is a code-obfuscation resilient work that searches for the semantical difference of the original and the suspicious binary. However, `CoP` approximates the similarity between the binaries and reports a score indicating the likelihood that the original binary's components will be reused. This is different from our work as we check and prove the functional equivalence between the binaries.

Learning-based analysis. In recent years, there has been significant research on applying learning-based techniques such as machine learning [30,37,42], deep neural networks [33, 48], and natural language processing [50] for similarity detection. Although learning-based methods are efficient and show promising results, they require extraneous training of an available dataset. Moreover, preprocessing necessary information for different diversification techniques is difficult due to the uncertainty of target modification. Hence, learning-based binary similarity detection is not yet resilient concerning code diversification techniques.

Low-level Formal Verification. Formal verification of low-level code (e.g., assembly Language) has been an active research field for decades [8, 49]. CompCert [29] is a formally verified compiler which provides the guarantee that safety properties proved on the source code hold for executable. However, to the best of our knowledge, CompCert cannot be used for binary diversification.

Address Space Layout Randomization. Address Space Layout Randomization (ASLR) is a widely deployed code diversification technique. ASLR randomly arranges the addresses of various parts of a process without major modifications to the actual binary; instead, the operating system ensures diversification at execution time. In other words, it executes the *same* binary in different ways. Our methodology thus does not apply to ASLR, as our methodology is targeted towards establishing equivalence between two *di erent* binaries.

7 Conclusion

Code diversification is a security technique that produces multiple binaries that are different but semantically equal. This paper presents the first scalable and automated technique to establish the soundness of a diversification tool. Soundness is expressed by establishing an equivalence relation between vanilla and diversified binary. The technique is based on disassembly, symbolic execution, establishing stack-pointer related invariants, and establishing mappings between memory regions in both the vanilla and the diversified world. Overall, our work provides proof of semantical equivalence between roughly 87% and 96% of the text sections in the binaries. Our work's main limitation is indirect branching (e.g., indirect jumps) due to dynamically computed addresses. We aim to resolve these limitations for future work.

Acknowledgement

Project information can be found at: https://llrm-project.org/. All source codes and scripts are available at: https://github.com/jjang3/NFM_2021 This work is supported in part by the US Office of Naval Research (ONR) under grant N00014-17-1-2297 and NSWCDD/NEEC under grant N00174-20-1-0009.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
2. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a board range of memory error exploits. In: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12. p. 8. SSYM'03, USENIX Association, USA (2003)
3. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14. p. 17. SSYM'05, USENIX Association, USA (2005)
4. Browne, M., Clarke, E., Grümberg, O.: Characterizing finite kripke structures in propositional temporal logic. Theoretical Computer Science **59**(1), 115–131 (1988), https://doi.org/10.1016/0304-3975(88)90098-9
5. Chae, D.K., Ha, J., Kim, S.W., Kang, B., Im, E.G.: Software plagiarism detection: A graph-based approach. In: Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management. pp. 1577–1580. CIKM '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2505515.2507848
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 559–572. CCS '10, ACM, New York, NY, USA (2010), https://doi.acm.org/10.1145/1866307.1866370
7. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1027–1040. PLDI 2019, ACM, New York, NY, USA (2019), https://doi.acm.org/10.1145/3314221.3314596
8. Clutterbuck, D.L., Carré, B.A.: The verification of low-level code. Softw. Eng. J. **3**(3), 97–111 (May 1988), https://dx.doi.org/10.1049/sej.1988.0012
9. Cohen, F.B.: Operating system protection through program evolution. vol. 12, p. 565–584. Elsevier Advanced Technology Publications, GBR (Oct 1993), https://doi.org/10.1016/0167-4048(93)90054-9
10. Crane, S., Homescu, A., Larsen, P.: Code randomization: Haven't we solved this problem yet? In: 2016 IEEE Cybersecurity Development (SecDev). pp. 124–129 (2016)
11. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.R., Brunthaler, S., Franz, M.: Readactor: Practical code randomization resilient to memory disclosure. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. pp. 763–780. SP '15, IEEE Computer Society, Washington, DC, USA (2015), https://doi.org/10.1109/SP.2015.52
12. David, Y., Partush, N., Yahav, E.: Statistical similarity of binaries. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 266–280. PLDI '16, ACM, New York, NY, USA (2016), https://doi.acm.org/10.1145/2908080.2908126
13. Egele, M., Woo, M., Chapman, P., Brumley, D.: Blanket execution: Dynamic similarity testing for program binaries and components. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 303–317. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

14. Fernandez, J.C., Mounier, L.: Verifying bisimulations “on the fly”. In: FORTE. vol. 90, pp. 95–110 (1990)
15. Forrest, S., Somayaji, A., Ackley, D.: Building diverse computer systems. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI). pp. 67–. HOTOS ’97, IEEE Computer Society, Washington, DC, USA (1997), http://dl.acm.org/citation.cfm?id=822075.822408
16. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs pp. 238–255 (2008)
17. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: Proceedings of the 21st USENIX Conference on Security Symposium. p. 40. Security’12, USENIX Association, USA (2012)
18. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: Ilr: Where’d my gadgets go? In: 2012 IEEE Symposium on Security and Privacy. pp. 571–585 (2012)
19. Hiser, J., Nguyen-Tuong, A., Hawkins, W., McGill, M., Co, M., Davidson, J.: Zipr++: Exceptional binary rewriting. In: Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation. p. 9–15. FEAST ’17, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3141235.3141240
20. Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., Franz, M.: Profile-guided automated software diversity. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–11. CGO ’13, IEEE Computer Society, Washington, DC, USA (2013), https://doi.org/10.1109/CGO.2013.6494997
21. Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J.M., Holvitie, J., Hyrynsalmi, S., Leppänen, V.: Diversification and obfuscation techniques for software security: A systematic literature review. Information and Software Technology **104**, 72 – 93 (2018)
22. Jackson, T., Homescu, A., Crane, S., Larsen, P., Brunthaler, S., Franz, M.: Diversifying the software stack using randomized nop insertion. In: Jajodia, S., Ghosh, A.K., Subrahmanian, V.S., Swarup, V., Wang, C., Wang, X.S. (eds.) Moving Target Defense, Advances in Information Security, vol. 100, pp. 151–173. Springer (2013)
23. Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M.: Compiler-Generated Software Diversity, pp. 77–98. Springer New York, New York, NY (2011)
24. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM – software protection for the masses. In: Wyseur, B. (ed.) Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO’15, Firenze, Italy, May 19th, 2015. pp. 3–9. IEEE (2015). https://doi.org/10.1109/SPRO.2015.10
25. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC’06). pp. 339–348 (2006)
26. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Proceedings of the 8th International Symposium on Static Analysis. pp. 40–56. SAS ’01, Springer-Verlag, London, UK, UK (2001), http://dl.acm.org/citation.cfm?id=647170.718283
27. Koo, H., Chen, Y., Lu, L., Kemerlis, V.P., Polychronakis, M.: Compiler-assisted code randomization. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 461–477 (May 2018). https://doi.org/10.1109/SP.2018.00029

28. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: 2014 IEEE Symposium on Security and Privacy. pp. 276–291 (May 2014). https://doi.org/10.1109/SP.2014.25
29. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009), http://doi.acm.org/10.1145/1538788.1538814
30. Li, L., Feng, H., Zhuang, W., Meng, N., Ryder, B.: Cclearner: A deep learning-based clone detection approach. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 249–260 (Sep 2017). https://doi.org/10.1109/ICSME.2017.46
31. Liang, Y., Ma, X., Wu, D., Tang, X., Gao, D., Peng, G., Jia, C., Zhang, H.: Stack layout randomization with minimal rewriting of android binaries. In: Kwon, S., Yun, A. (eds.) Information Security and Cryptology - ICISC 2015. pp. 229–245. Springer International Publishing, Cham (2016)
32. Lim, J.P., Nagarakatte, S.: Automatic equivalence checking for assembly implementations of cryptography libraries pp. 37–49 (2019), http://dl.acm.org/citation.cfm?id=3314872.3314880
33. Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W.: αdiff: Cross-version binary code similarity detection with dnn. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 667–678. ASE 2018, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3238147.3238199
34. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: Detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 872–881. KDD '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1150402.1150522
35. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 389–400. FSE 2014, ACM, New York, NY, USA (2014), https://doi.acm.org/10.1145/2635868.2635900
36. Pappas, V., Polychronakis, M., Keromytis, A.D.: Practical software diversification using in-place code randomization. In: Moving Target Defense (2013)
37. Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., Jin, Z.: Building program vector representations for deep learning. In: Zhang, S., Wirsing, M., Zhang, Z. (eds.) Knowledge Science, Engineering and Management. pp. 547–553. Springer International Publishing, Cham (2015)
38. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy. pp. 709–724 (May 2015). https://doi.org/10.1109/SP.2015.49
39. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 669–685. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032305.2032360
40. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (May 2009), http://dx.doi.org/10.1016/j.scico.2009.02.007
41. Shacham, H.: 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. pp. 552–561. CCS '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1315245.1315313

42. Shalev, N., Partush, N.: Binary similarity detection using machine learning. In: Proceedings of the 13th Workshop on Programming Languages and Analysis for Security. pp. 42–47. PLAS '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3264820.3264821
43. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016 IEEE Symposium on Security and Privacy (SP) pp. 138–157 (2016)
44. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. pp. 157–168. ISSTA '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1146238.1146256
45. Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., Vigna, G.: Ramblr: Making reassembly great again. In: In The Network and Distributed System Security Symposium. NDSS '17 (2017). https://doi.org/10.14722/ndss.2017.23225
46. Wang, S., Wang, P., Wu, D.: Composite software diversification. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 284–294 (2017)
47. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. p. 157–168. CCS '12, Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2382196.2382216
48. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 363–376. CCS '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3133956.3134018
49. Xu, Z., Miller, B.P., Reps, T.: Safety checking of machine code. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. pp. 70–82. PLDI '00, ACM, New York, NY, USA (2000), http://doi.acm.org/10.1145/349299.349313
50. Zuo, F., Li, X., Zhang, Z., Young, P., Luo, L., Zeng, Q.: Neural machine translation inspired binary code similarity comparison beyond function pairs. CoRR **abs/1808.04706** (2018), http://arxiv.org/abs/1808.04706