Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks

Rodothea Myrsini Tsoupidi[1], Roberto Castañeda Lozano[2], and Benoit Baudry[1]

[1] KTH Royal Institute of Technology, Sweden
{tsoupidi,baudry}@kth.se
[2] University of Edinburgh, United Kingdom
roberto.castaneda@ed.ac.uk

Abstract. Modern software deployment process produces software that is uniform, and hence vulnerable to large-scale code-reuse attacks. *Compiler-based diversification* improves the resilience and security of software systems by automatically generating different assembly code versions of a given program. Existing techniques are efficient but do not have a precise control over the quality of the generated code variants.
This paper introduces *Diversity by Construction (DivCon)*, a constraint-based compiler approach to software diversification. Unlike previous approaches, DivCon allows users to control and adjust the conflicting goals of diversity and code quality. A key enabler is the use of Large Neighborhood Search (LNS) to generate highly diverse assembly code efficiently. Experiments using two popular compiler benchmark suites confirm that there is a trade-off between quality of each assembly code version and diversity of the entire pool of versions. Our results show that DivCon allows users to trade between these two properties by generating diverse assembly code for a range of quality bounds. In particular, the experiments show that DivCon is able to mitigate code-reuse attacks effectively while delivering near-optimal code ($< 10\%$ optimality gap).
For constraint programming researchers and practitioners, this paper demonstrates that LNS is a valuable technique for finding diverse solutions. For security researchers and software engineers, DivCon extends the scope of compiler-based diversification to performance-critical and resource-constrained applications.

Keywords: compiler-based software diversification · code-reuse attacks · constraint programming · embedded systems

1 Introduction

Good software development practices, such as code reuse [19], continuous deployment, and automatic updates contribute to the emergence of software monocultures [3]. While such monocultures facilitate software distribution, bug reporting, and software authentication, they also introduce serious risks related to the wide spreading of attacks against all users that run identical software.

Software diversification is a method to mitigate the problems caused by uniformity. Similarly to biodiversity, software diversification improves the resilience and security of a software system [2] by introducing diversity in it. Software diversification can be applied in different phases of the software development cycle, i.e. during implementation, compilation, loading, execution, and more [20]. This paper is concerned with *compiler-based* diversification, which automatically generates different assembly code versions from a single source program.

Modern compilers do not merely aim to generate correct code, but also code that is of high quality. Existing compiler-based diversification techniques are efficient and effective at diversifying assembly code [20] but do not have a precise control over its quality and may produce unsatisfactory results. These techniques (discussed in Section 5) are either based on randomizing heuristics or in high-level superoptimization methods that do not capture accurately the quality of the generated code.

This paper introduces Diversity by Construction (DivCon), a compiler-based diversification approach that allows users to control and adjust the conflicting goals of quality of each code version and diversity among all versions. DivCon uses a Constraint Programming (CP)-based compiler backend to generate multiple solutions corresponding to functionally equivalent program variants according to an accurate code quality model. The backend models the input program, the hardware architecture, and the compiler transformations as a constraint problem, whose solution corresponds to assembly code for the input program.

The use of CP makes it possible to 1) control the quality of the generated solutions by constraining the objective function, 2) introduce application-specific constraints that restrict the diversified solutions, and 3) apply sophisticated search procedures that are particularly suitable for diversification. In particular, DivCon uses Large Neighborhood Search (LNS) [29], a popular metaheuristic in multiple application domains, to generate highly diverse solutions efficiently.

Our experiments compiling 17 functions from two popular compiler benchmark suites to the MIPS32 architecture confirm that there is a trade-off between code quality and diversity, and demonstrate that DivCon allows users to navigate this conflict by generating diverse assembly code for a range of quality bounds. In particular, the experiments show that DivCon is able to mitigate code-reuse attacks effectively while guaranteeing a code quality of 10% within optimality.

For constraint programming researchers and practitioners, this paper demonstrates that LNS is a valuable technique for finding diverse solutions. For security researchers and software engineers, DivCon extends the scope of compiler-based diversification to performance-critical and resource-constrained applications, and provides a solid step towards secure-by-construction software.

Contributions. To summarize, this paper:

- proposes a CP-based technique for compiler-based, quality-aware software diversification (Section 3);
- shows that LNS is a promising technique for generating highly diverse solutions efficiently (Section 4.3);

```
1  0x9d001408: ...
2  0x9d00140c: lw    $s2, 4($sp)
3  0x9d001410: lw    $s4, 0($sp)
4  0x9d001414: jr    $t9
5  0x9d001418: addiu $sp, $sp, 16
```

(a) Original gadget.

```
1  0x9d001408: lw    $s2, 4($sp)
2  0x9d00140c: nop
3  0x9d001410: lw    $s4, 0($sp)
4  0x9d001414: jr    $t9
5  0x9d001418: addiu $sp, $sp, 16
```

(b) Diversified gadget.

Fig. 1: Example gadget diversification in MIPS32 assembly code

– quantifies the trade-off between code quality and diversity (Section 4.4); and
– demonstrates that DivCon mitigates code-reuse attacks effectively while preserving high code quality (Section 4.5).

2 Background

This section describes code-reuse attacks (Section 2.1), diversification approaches in CP (Section 2.2), and combinatorial compiler backends (Section 2.3).

2.1 Code-reuse Attacks

Code-reuse attacks take advantage of memory vulnerabilities, such as buffer overflows, to reuse program code for malicious purposes. More specifically, code-reuse attacks insert data into the program memory to affect the control flow of the program and execute code that is valid but unintented.

Jump-Oriented Programming (JOP)[3] is a code-reuse attack [7,4] that combines different code snippets from the original program code to form a Turing complete language for attackers. These code snippets terminate with a branch instruction. The building blocks of a JOP attack are *gadgets*: meta-instructions that consist of one or multiple code snippets and have specific semantics. Figure 1a shows a JOP gadget found by the *ROPgadget* tool [27] in a MIPS32 binary. Assuming that the attacker controls the stack, lines 2 and 3 load attacker data in registers $s2 and $s4, respectively. Then, line 4 jumps to the address of register $t9. The last instruction (line 5) is placed in a delay slot and, hence, it is executed before the jump [31]. The semantics of this gadget depends on the actual attack payload and might be to load a value to register $s2 and $s4. Then, the program jumps to the next gadget that resides at the stack address of $t9.

Statically designed JOP attacks use the absolute binary addresses for installing the attack payload. Hence, a simple change in the instruction schedule of the program as in Figure 1b prevents a JOP attack designed for Figure 1a. An attacker that designs an attack based on the binary of the original program assumes the presence of a gadget (Figure 1a) at position 0x9d00140c. However, in the diversified version, address 0x9d00140c does not start with the initial `lw`

[3] This paper focuses on JOP due to the characteristics of MIPS32, but could be generalized to other code-reuse attacks such as Return-Oriented Programming (ROP) [28].

instruction of Figure 1a, and by the end of the execution of the gadget, register \$s2 does not contain the attacker data. In this way, diversification can break the semantics of the gadget and mitigate an attack against the diversified code.

2.2 Diversity in Constraint Programming

While typical CP applications aim to discover either some solution or the optimal solution, some applications require finding *diverse* solutions for various purposes.

Hebrard *et al.* [13] introduce the MaxDiverse*k*Set problem, which consists in finding the most diverse set of k solutions, and propose an exact and an incremental algorithm for solving it. The exact algorithm does not scale to a large number of solutions [32,16]. The incremental algorithm selects solutions iteratively by solving a distance maximization problem.

Automatic Generation of Architectural Tests (ATGP) is an application of CP that requires generating many diverse solutions. Van Hentenryck *et al.* [32] model ATGP as a MaxDiverse*k*Set problem and solve it using the incremental algorithm of Hebrard et. al. Due to the large number of diverse solutions required (50-100), Van Hentenryck *et al.* replace the maximization step with local search.

In software diversity, solution quality is of paramount importance. In general, earlier CP approaches to diversity are concerned with satisfiability only. An exception is the approach of Petit *et al.* [26]. This approach modifies the objective function for assessing both solution quality and solution diversity, but does not scale to the large number of solutions required by software diversity. Ingmar *et al.* [16] propose a generic framework for modeling diversity in CP. For tackling the quality-diversity trade-off, they propose constraining the objective function with the optimal (or best known) cost o. DivCon applies this approach by allowing solutions p% worse than o, where p is configurable.

2.3 Compiler Optimization as a Combinatorial Problem

A Constraint Satisfaction Problem (CSP) is a problem specification $P = \langle V, U, C \rangle$, where V are the problem variables, U is the domain of the variables, and C the constraints among the variables. A Constraint Optimization Problem (COP), $P = \langle V, U, C, O \rangle$, consists of a CSP and an objective function O. The goal of a COP is to find a solution that optimizes O.

Compilers are programs that generate low-level assembly code, typically optimized for *speed* or *size*, from higher-level source code. A compilation process can be modeled as a COP by letting V be the decisions taken during the translation, C be the constraints imposed by the program semantics and the hardware resources, and O be the cost of the generated code.

Compiler backends generate low-level assembly code from an Intermediate Representation (IR), a program representation that is independent of both the source and the target language. Figure 2 shows the high-level view of a *combinatorial* compiler backend. A combinatorial compiler backend takes as input the IR of a program, generates and solves a COP, and outputs the optimized low-level assembly code described by the solution to the COP.

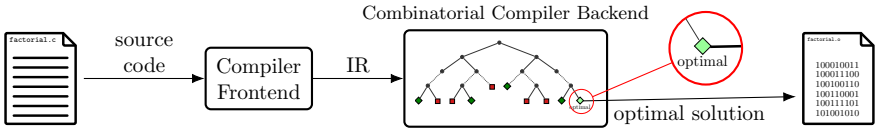


Fig. 2: High-level view of a combinatorial compiler backend

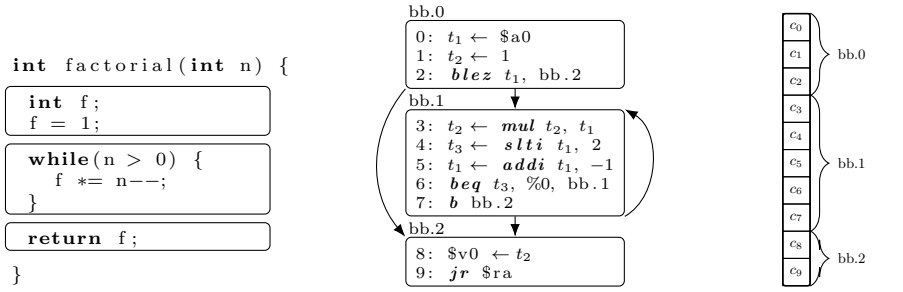
This paper assumes that programs at the IR level are represented by their Control-Flow Graph (CFG). A CFG is a representation of the possible execution paths of a program, where each node corresponds to a *basic block* and edges correspond to intra-block jumps. A *basic block*, in its turn, is a set of abstract instructions (hereafter just *instructions*) with no branches besides the end of the block. Each instruction is associated with a set of operands characterizing its input and output data. Typical decision variables V of a combinatorial compiler backend are the issue cycle $c_i \in \mathbb{N}_0$ of each instruction i, the processor instruction $m_i \in \mathbb{N}_0$ that implements each instruction i, and the processor register $r_o \in \mathbb{N}_0$ assigned to each operand o.

DivCon aims at mitigating code-reuse attacks. Therefore, DivCon considers the order of the instructions in the final binary, which directly affects the feasibility of code-reuse attacks (see Figures 1a and 1b). For this reason, the diversification model uses the issue cycle sequence of instructions, $c = \{c_0, c_1, ..., c_n\}$, to characterize the diversity among different solutions.

Figure 3a shows an implementation of the factorial function in C where each basic block is highlighted. Figure 3b shows the IR of the program. The example IR contains 10 instructions in three basic blocks: bb.0, bb.1, and bb.2. bb.0 corresponds to initializations, where `$a0` holds the function argument `n` and t_1 corresponds to variable `f`. bb.1 computes the factorial in a loop by accumulating the result in t_1. bb.2 stores the result to `$v0` and returns. Some instructions in the example are interdependent, which leads to serialization of the instruction schedule. For example, ***beq*** (6) consumes data (t_3) defined by ***slti*** (4) and hence needs to be scheduled later. Instruction dependencies limit the amount of possible assembly code versions and can restrict diversity significantly, as seen in Section 4.3. Finally, Figure 3c shows the arrangement of the issue cycle variables in the constraint model used by the combinatorial compiler backend.

3 DivCon

This section introduces DivCon, a software diversification method that uses a combinatorial compiler backend to generate program variants. Figure 4 shows a high-level view of the diversification process. DivCon uses 1) the optimal solution to start the *search* for diversification and 2) the cost of the optimal solution to restrict the variants within a maximum gap from the optimal. Subsequently, DivCon generates a number of solutions to the CSP that correspond to diverse program variants.



(a) C code (b) IR (c) Issue cycles

Fig. 3: Factorial function example

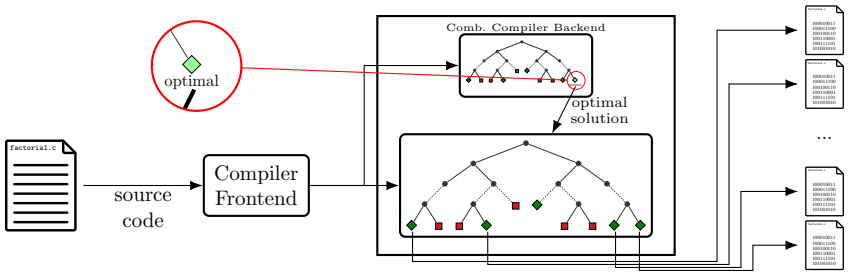


Fig. 4: High-level view of DivCon

The rest of this section describes the diversification approach of DivCon. Section 3.1 formulates the diversification problem in terms of the constraint model of a combinatorial compiler backend, Section 3.2 defines the distance measures, and finally, Section 3.3 describes the search strategy for generating program variants.

3.1 Problem Description

Let $P = \langle V, U, C \rangle$ be the compiler backend CSP for the program under compilation, O be the objective function, and o be the cost of the optimal or best known solution to the COP, $\langle V, U, C, O \rangle$. Let δ be a function that measures the distance between two solutions of P (two such functions are defined in Section 3.2). Let $h \in \mathbb{N}$ be the minimum pairwise distance and $p \in \mathbb{R}_{\geq 0}$ be the maximum optimality gap specified by the user. Our problem is to find a subset of the solutions to the CSP, $S \subseteq sol(P)$, such that $\forall s_1, s_2 \in S \,.\, s_1 \neq s_2 \implies \delta(s_1, s_2) \geq h$ and $\forall s \in S \,.\, O(s) \leq (1 + p) \cdot o$.

To solve the above problem, DivCon employs the incremental algorithm listed in Algorithm 1. Starting with the optimal solution y_{opt}, the algorithm adds the distance constraint for y_{opt} and the optimality constraint with $o = y_{opt}(O)$ (line

2). Notation $\delta(y)$ is used instead of $\delta(y, s) \,|\, \forall s \in sol(\langle V, U, C' \rangle)$ for readability. While the termination condition is not fulfilled (line 3), the algorithm uses LNS as described in Section 3.3 to find the next solution y (line 4), adds the next solution to the solution set S (line 5), and updates the distance constraints based on the latest solution (line 6). When the termination condition is satisfied, the algorithm returns the set of solutions S corresponding to diversified assembly code variants.

Algorithm 1: Incremental algorithm for generating diverse solutions

```
1  S ← {y_opt}, y ← y_opt,
2  C' ← C ∪ {δ(y_opt) ≥ h, O(V) ≤ (1 + p) · o}
3  while not term_cond() // e.g. |S| > k ∨ time_limit()
4      y ← solve_LNS(relax(y), ⟨V, U, C'⟩)
5      S ← S ∪ {y}
6      C' ← C' ∪ {δ(y) ≥ h}
```

Figure 5 shows two MIPS32 variants of the factorial example (Figure 3), which correspond to two solutions of DivCon. The variants differ in two aspects: first, the ***beqz*** instruction is issued one cycle later in Figure 5b than in Figure 5a, and second, the temporary variable t_3 (see Figure 3) is assigned to different MIPS32 registers ($t0 and $t1).

3.2 Distance Measures

This section defines two alternative distance measures: Hamming Distance (HD) and Levenshtein Distance (LD). Both distances operate on the schedule of the instructions, i.e. the order in which the instructions are issued in the CPU.

Hamming Distance (HD). HD is the Hamming distance [12] between the issue cycle variables of two solutions. Given two solutions $s, s' \in sol(P)$:

$$\delta_{HD}(s, s') = \sum_{i=0}^{n} (s(c_i) \neq s'(c_i)), \tag{1}$$

```
1  bb.0:  blez  $a0, bb.2
2         addiu $v0, $zero, 1
3  bb.1:  mul   $v0, $v0, $a0
4         slti  $t0, $a0, 2
5         beqz  $t0, bb.1
6         addi  $a0, $a0, -1
7  bb.2:  jr    $ra
8         nop
```

(a) Variant 1.

```
1  bb.0:  blez  $a0, bb.2
2         addiu $v0, $zero, 1
3  bb.1:  mul   $v0, $v0, $a0
4         slti  $t1, $a0, 2
5         nop
6         beqz  $t1, bb.1
7         addi  $a0, $a0, -1
8  bb.2:  jr    $ra
9         nop
```

(b) Variant 2.

Fig. 5: Two MIPS32 variants of the factorial example in Figure 3

where n is the maximum number of instructions.

Consider Figure 1b, a diversified version of the gadget in Figure 1a. The only instruction that differs from Figure 1a is the instruction at line 1 that is issued one cycle before. The two examples have a HD of one, which in this case is enough for breaking the functionality of the original gadget (see Section 2.1).

Levenshtein Distance (LD). LD (or edit distance) measures the minimum number of edits, i.e. insertions, deletions, and replacements, that are necessary for transforming one instruction schedule to another. Compared to HD, which considers only *replacements*, LD also considers *insertions* and *deletions*. To understand this effect, consider Figure 5. The two gadgets differ only by one `nop` operation but HD gives a distance of three, whereas LD gives one, which is more accurate. LD takes ordered vectors as input, and thus requires an ordered representation (as opposed to a detailed schedule) of the instructions. Therefore, LD uses vector $c^{-1} = channel(c)$, a sequence of instructions ordered by their issue cycle. Given two solutions $s, s' \in sol(P)$:

$$\delta_{LD}(s, s') = \texttt{levenshtein\_distance}(s(c^{-1}), s'(c^{-1})), \tag{2}$$

where `levenshtein_distance` is the WagnerFischer algorithm [33] with time complexity $O(nm)$, where n and m are the lengths of the two sequences.

3.3 Search

Unlike previous CP approaches to diversity, DivCon employs Large Neighborhood Search (LNS) for diversification. LNS is a metaheuristic that defines a neighborhood, in which *search* looks for better solutions, or, in our case, different solutions. The definition of the neighborhood is through a *destroy* and a *repair* function. The *destroy* function unassigns a subset of the variables in a given solution and the *repair* function finds a new solution by assigning new values to the *destroyed* variables.

In DivCon, the algorithm starts with the optimal solution of the combinatorial compiler backend. Subsequently, it destroys a part of the variables and continues with the model's branching strategy to find the next solution, applying a restart after a given number of failures. LNS uses the concept of *neighborhood*, i.e. the variables that LNS may destroy at every restart. To improve diversity, the neighborhood for DivCon consists of all decision variables, i.e. the issue cycles c, the instruction implementations m, and the registers r. Furthermore, LNS depends on a *branching strategy* to guide the *repair* search. To improve security and allow LNS to select diverse paths after every restart, DivCon employs a random variable-value selection branching strategy as described in Table 1b.

4 Evaluation

The evaluation of DivCon addresses four main questions:

- RQ1. What is the scalability of the distance measures in generating multiple program variants? Here, we evaluate which of the distance measures is the most appropriate for software diversification.
- RQ2. How effective and how scalable is LNS for code diversification? Here, we investigate LNS as an alternative approach to diversity in CP.
- RQ3. How does code quality relate to code diversity and what are the involved trade-offs?
- RQ4. How effective is DivCon at mitigating code-reuse attacks? This question is the main application of CP-based diversification in this work.

4.1 Experimental Setup

Implementation. DivCon is implemented as an extension of Unison [6], and is available at `https://github.com/romits800/divcon`. Unison implements two backend transformations: instruction scheduling and register allocation. DivCon employs Unison's solver portfolio including Gecode v6.2 [11] and Chuffed v0.10.3 [8] to find optimal solutions, and Gecode v6.2 only for diversification. The LLVM compiler [21] is used as a front-end and IR-level optimizer.

Benchmark functions and platform. The evaluation uses 17 functions sampled randomly from MediaBench [22] and SPEC CPU2006 [30], two benchmark suites widely employed in embedded and general-purpose compiler research. The size of the functions is limited to between 10 and 30 instructions (with a median of 20 instructions) to keep the evaluation of all methods and distance measures feasible regardless of their computational cost. Table 2 lists the ID, application, name, basic blocks (b), and instructions (i) of each sampled function. The functions are compiled to MIPS32 assembly code. MIPS32 is a popular architecture within embedded systems and the security-critical Internet of Things [1].

Host platform. All experiments run on an Intel®Core™i9-9920X processor at 3.50GHz with 64GB of RAM running Debian GNU/Linux 10 (buster). Each of the experiments runs for 20 random seeds. The results show the mean value and the standard deviation from these experiments. The available virtual memory for each of the executions is 10GB. The experiments for different random seeds run in parallel (5 seeds at a time), with two unique cores available for every seed for overheating reasons. DivCon runs as a sequential program.

Table 1: ORIGINAL and RANDOM branching strategies

(a) ORIGINAL branching strategy.

Variable	Var. Selection	Value Selection
c_i	in order	min. val first
m_i	in order	min. val first
r_o	in order	randomly

(b) RANDOM branching strategy.

Variable	Var. Selection	Value Selection
c_i	randomly	randomly
m_i	randomly	randomly
r_o	randomly	randomly

Table 2: Benchmark functions

ID	app	function name	b	i
b1	sphinx3	ptmr_init	1	10
b2	gcc	ceil_log2	1	14
b3	mesa	glIndexd	1	14
b4	h264ref	symbol2uvlc	1	15
b5	gobmk	autohelperowl_defen..	1	23
b6	mesa	glVertex2i	1	23
b7	hmmer	AllocFancyAli	1	25
b8	gobmk	autohelperowl_vital..	1	27
b9	gobmk	autohelperpat1088	1	29
b10	gobmk	autohelperowl_attac..	1	30
b11	gobmk	get_last_player	3	13
b12	h264ref	UpdateRandomAccess	3	16
b13	gcc	xexit	3	17
b14	gcc	unsigned_condition	3	24
b15	sphinx3	glist_tail	4	10
b16	gcc	get_frame_alias_set	5	20
b17	gcc	parms_set	5	25

Table 3: Scalability of δ_{HD}, δ_{LD}

ID	δ_{HD}		δ_{LD}	
	$t(s)$	num	$t(s)$	num
b1	0.1±0.2	26	131.2±131.4	26
b2	1.0±0.1	200	-	68
b3	1.1±0.1	200	-	58
b4	0.7±0.0	200	-	73
b5	2.3±0.3	200	-	38
b6	2.5±0.2	200	-	35
b7	2.0±0.3	200	-	37
b8	3.8±0.8	200	-	35
b9	4.0±0.6	200	-	28
b10	4.5±0.7	200	-	27
b11	1.3±0.1	200	-	56
b12	1.1±0.2	200	-	47
b13	0.8±0.1	200	-	91
b14	1.8±0.3	200	-	27
b15	1.7±0.2	200	-	60
b16	2.7±0.4	200	-	31
b17	1.6±0.2	200	-	35

Algorithm Configuration. The experiments focus on speed optimization and aim to generate 200 variants within a timeout. Parameter h in Algorithm 1 is set to one because even small distance between variants is able to break gadgets (see Figure 1). LNS uses restart-based search with a limit of 500 failures, and a relax rate of 70%. The *relax rate* is the probability that LNS destroys a variable at every restart, which affects the distance between two subsequent solutions. A higher relax rate increases diversity but requires more solving effort. We have found experimentally that 70% is an adequate balance between the two. All experiments are available at `https://github.com/romits800/divcon_experiments`.

4.2 RQ1. Scalability of the Distance Measures

The ability to generate a large number of variants is paramount for software diversification. This section compares the distance measures introduced in Section 3.2 with regards to scalability.

Table 3 presents the results of the distance evaluation, where a time limit of 10 minutes and optimality gap of $p = 10\%$ are used. For each distance measure (δ_{HD} and δ_{LD}) the table shows the diversification time t, in seconds (or "-" if the algorithm is not able to generate 200 variants) and the number of generated variants *num* within the time limit.

The results show that for δ_{HD}, DivCon is able to generate 200 variants for all benchmarks except *b1*, which has exactly 26 variants. The diversification time for δ_{HD} is less than 5 seconds for all benchmarks. Distance δ_{LD}, on the other hand, is not able to generate 200 variants for any of the benchmarks within the time limit. This poor scalability of δ_{LD} is due to the quadratic complexity of its implementation [33], whereas HD can be implemented linearly. Consequently, the rest of the evaluation uses δ_{HD}.

4.3 RQ2. Scalability and Diversification Effectiveness of LNS

This section evaluates the diversification effectiveness and scalability of LNS compared to incremental MaxDiverse$k$Set (where the first solution is found randomly and the maximization step uses the branching strategy from Table 1a) and Random Search (RS) (which uses the branching strategy from Table 1b).

To measure the diversification effectiveness of these methods, the evaluation uses the relative pairwise distance of the solutions. Given a set of solutions S and a distance measure δ, the pairwise distance d of the variants in S is $d(\delta, S) = \sum_{i=0}^{|S|} \sum_{j>i}^{|S|} \delta(s_i, s_j) \,/\, \binom{|S|}{2}$. The *larger* this distance, the more diverse the solutions are, and thus, diversification is more effective. Table 4 shows the pairwise distance d and diversification time t for each benchmark and method, where the experiment uses a time limit of 30 minutes and optimality gap of $p = 10\%$. The best values of d (larger) and t (lower) are marked in **bold** for the completed experiments, whereas incomplete experiments are highlighted in *italic* and their number of variants in parenthesis.

Table 4: Distance and Scalability of LNS with RS and MaxDiverse$k$Set

ID	MaxDiverse$k$Set		RS		LNS (0.7)	
	d	$t(s)$	d	$t(s)$	d	$t(s)$
b1	*4.1±0.0*	0.2±0.0 (26)	*4.1±0.0*	**0.0±0.0 (26)**	*4.1±0.0*	0.1±0.2 (26)
b2	**10.8±0.0**	761.8±10.1	6.4±0.2	**0.6±0.1**	8.6±0.6	1.0±0.1
b3	*14.6±0.0*	- (21)	5.8±0.1	**0.6±0.1**	**10.8±0.8**	1.0±0.1
b4	*14.4±0.0*	- (19)	4.3±0.1	**0.2±0.0**	**12.1±0.3**	0.6±0.0
b5	*22.0±0.0*	- (2)	4.3±0.3	**0.5±0.0**	**16.1±1.1**	2.2±0.3
b6	*22.9±0.4*	- (2)	5.3±0.0	**1.0±0.1**	**16.4±0.6**	2.4±0.2
b7	*24.9±0.1*	- (6)	4.5±0.2	**0.4±0.0**	**18.1±1.2**	1.9±0.3
b8	*24.8±0.4*	- (2)	6.5±0.2	**3.5±0.5**	**17.2±0.9**	3.8±0.8
b9	*26.0±0.0*	- (2)	4.2±0.3	**0.4±0.0**	**19.8±0.7**	3.9±0.6
b10	*28.0±0.0*	- (2)	6.0±0.0	5.3±1.0	**20.1±1.1**	**4.5±0.7**
b11	**13.8±0.0**	356.9±8.2	5.3±0.1	**0.2±0.0**	10.1±1.0	1.2±0.1
b12	*21.5±0.1*	- (5)	6.4±0.9	**0.2±0.0**	**14.9±1.0**	1.0±0.2
b13	*17.4±0.0*	- (122)	6.7±0.0	0.9±0.1	**12.0±0.9**	**0.7±0.1**
b14	*30.1±0.0*	- (20)	7.5±0.2	**0.2±0.0**	**24.9±0.7**	1.8±0.3
b15	-	-	2.6±0.3	**0.1±0.0**	**20.2±0.5**	1.6±0.2
b16	-	-	5.6±0.4	**0.3±0.0**	**21.3±0.8**	2.6±0.4
b17	-	-	*2.9±0.1*	- (91)	**28.1±1.5**	**1.6±0.2**

The scalability results ($t(s)$) show that RS and LNS are scalable (generate the maximum 200 variants for almost all benchmarks), whereas MaxDiverse$k$Set scales poorly (cannot generate 200 variants for any benchmark but *b2* and *b11*). Both *b2* and *b11* have a small search space (few, highly interdependent instructions), which leads to restricted diversity but facilitates solving. For *b1*, all instructions are interdependent on each other, which forces a linear schedule and results in only 26 possible variants (given $p = 10\%$). On the other end, MaxDiverse$k$Set is not able to find any variants for *b15*, *b16*, and *b17*. These benchmarks have many basic blocks resulting in a more complex objective function. For the largest benchmark (*b17*), only LNS is able to scale up to 200 solutions. LNS is generally slower than RS, but for both LNS and RS all benchmarks have a diversification time less than six seconds.

The diversity results (d) show that LNS is more effective at diversifying than RS. The improvement of LNS over RS ranges from 35% (for *b2*) to 675% (for

b15). In the two cases where MaxDiverse$k$Set terminates (benchmarks *b2* and *b11*), it generates the most diverse code, as can be expected.

In summary, LNS offers an attractive balance between scalability and diversification effectiveness: it is close in scalability to, and sometimes improves, the overly fastest method (RS), but it is significantly and consistently more effective at diversifying code.

4.4 RQ3. Trade-off Between Code Quality and Diversity

A key advantage of using a CP-based compiler approach for software diversity is the ability to control the quality of the generated solutions. This ability enables control over the relation between the quality of each individual solution and the diversity of the entire pool of solutions. Insisting in optimality limits the number of possible diversified variants and their pairwise distance, whereas relaxing optimality allows higher diversity.

Table 5 shows the pairwise distance d (defined in Section 4.3), and the number of generated variants *num*, for all benchmarks and different values of the optimality gap $p \in \{0\%, 5\%, 10\%, 20\%\}$. LNS is used with a time limit of 10 minutes. The best values of d are marked in **bold**.

Table 5: Solution diversity for different optimality gap values

ID	0%		5%		10%		20%	
	d	num	d	num	d	num	d	num
b1	-	-	-	-	*4.1±0.0*	26	**6.5±0.1**	200
b2	*3.5±0.0*	9	6.7±0.4	200	8.6±0.6	200	**10.0±0.8**	200
b3	7.0±0.1	200	9.4±0.5	200	10.8±0.8	200	**14.8±1.0**	200
b4	7.8±0.2	200	10.1±0.3	200	12.1±0.3	200	**14.0±0.2**	200
b5	8.4±0.1	200	11.9±0.7	200	16.1±1.1	200	**19.7±0.6**	200
b6	10.8±0.1	200	14.7±0.4	200	16.4±0.6	200	**20.9±0.8**	200
b7	11.3±0.3	200	13.8±0.7	200	18.1±1.2	200	**22.8±1.1**	200
b8	11.0±0.1	200	13.6±0.6	200	17.2±0.9	200	**22.4±1.1**	200
b9	12.7±0.1	200	17.7±0.8	200	19.8±0.7	200	**24.4±0.6**	200
b10	13.7±0.1	200	18.1±0.9	200	20.1±1.1	200	**26.3±0.6**	200
b11	*2.0±0.0*	4	6.6±0.1	200	10.1±1.0	200	**14.2±0.9**	200
b12	*3.8±0.0*	10	10.3±1.2	200	14.9±1.0	200	**19.8±1.0**	200
b13	*2.1±1.3*	4	10.1±0.9	200	12.0±0.9	200	**15.7±1.2**	200
b14	*3.6±0.0*	24	21.0±0.6	200	24.9±0.7	200	**29.0±0.5**	200
b15	*2.4±0.0*	8	15.6±0.6	200	20.2±0.5	200	**23.5±1.4**	200
b16	*4.1±0.0*	44	15.1±1.1	200	21.3±0.8	200	**30.7±0.9**	200
b17	7.5±0.2	200	20.3±1.4	200	28.1±1.5	200	**38.4±0.9**	200

The first interesting observation is that even with no degradation of quality ($p = 0\%$), DivCon is able to generate a large number of variants for a significant fraction of the benchmarks. These include functions with a relatively large solution space, typically with a few large basic blocks where instructions are relatively independent of each other (*b3*-*b10* and *b17*). On the other hand, benchmarks with small basic blocks and many instruction dependencies (*b1*, *b2*, and *b11*-*b16*) provide fewer options for diversification, which results in a limited number of optimal variants.

Second, we observe that as soon as we slightly relax the constraint over optimality ($p = 5\%$), diversity radiates and DivCon generates 200 variants for all

benchmarks except *b1*. Then, the more we increase the optimality gap, the larger the diversification space grows and the distance between the variants increases. Table 5 illustrates one of the key contributions of DivCon: the ability to explore the trade-off between optimal solutions and highly diverse solutions.

In summary, depending on the characteristics of the compiled code, it is possible to generate a large number of variants without sacrificing optimality, and the code quality can be adjusted to further improve diversity if required by the targeted application.

4.5 RQ4. Code-Reuse Mitigation Effectiveness

Software Diversity has various applications in security, including mitigating code-reuse attacks. To measure the level of mitigation that DivCon achieves, we assess the gadget survival rate $srate(s_i, s_j)$ between two variants $s_i, s_j \in S$, where S is the set of generated variants. This metric determines how many of the gadgets of variant s_i appear at the same position on the other variant s_j, that is $srate(s_i, s_j) = |gad(s_i) - gad(s_j)| \,/\, |gad(s_i)|$, where $gad(s_i)$ are the gadgets in solution s_i. The procedure for computing $srate(s_i, s_j)$ is as follows: 1) run ROPgadget [27] to find the set of gadgets $gad(s_i)$ in solution s_i, and 2) for every $g \in gad(s_i)$, check whether there exists a gadget identical to g at the same address of s_j. This comparison is syntactic after removing all `nop` instructions.

This section compares the *srate* for all permutations of pairs in S, for all benchmarks, and for different values of the optimality gap using a time limit of 10 minutes. Low *srate* corresponds to higher mitigation effectiveness because code-reuse attacks based on gadgets in one variant have lower chances of locating the same gadgets in the other variants (see Figure 1).

Table 6 summarizes the gadget survival distribution for all benchmarks and different values of the optimality gap (0%, 5%, 10%, and 20%). Due to its skewness, the distribution of *srate* is represented as a histogram with four buckets (0%, (0%, 10%], (10%,40%], and (40%, 100%]) rather than summarized using common statistical measures. Here the best is a $srate(s_i, s_j)$ of 0%, which means that s_j does not contain any gadgets that exist in s_i, whereas a $srate(s_i, s_j)$ in range (40%,100%] means that s_j shares more than 40% of the gadgets of s_i. The values in **bold** correspond to the mode(s) of the histogram.

First, we notice that DivCon can generate some pairs of variants that share no gadget, even without relaxing the constraint of optimality ($p = 0\%$). This indicates that the pareto front of optimal code naturally includes software diversity that is good for security. Second, the results show that this effectiveness can be further increased by relaxing the constraint on code quality, with diminishing returns beyond $p = 10\%$. For $p = 0\%$, there are 10 benchmarks dominated by a 0% survival rate, whereas there are 7 benchmarks dominated by a weak $10\% - 40\%$-survival rate. The latter are still considered vulnerable to code-reuse attacks. However, increasing the optimality gap to just $p = 5\%$ makes 0% survival rate the dominating bucket for all benchmarks, and further increasing the gap to 10% and 20% increases significantly the number of pairs where no single

Table 6: Gadget survival rate for different optimality gap values

ID	0%					5%					10%					20%				
	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num
b1	-	-	-	-	-	-	-	-	-	-	**84**	3	3	10	26	**94**	4	2	1	200
b2	-	-	**69**	31	9	**60**	12	23	4	200	**76**	11	12	1	200	**81**	9	10	-	200
b3	**66**	15	18	1	200	**71**	14	15	1	200	**73**	13	13	1	200	**77**	14	9	-	200
b4	**94**	6	-	-	200	**96**	4	-	-	200	**96**	4	-	-	200	**98**	2	-	-	200
b5	**90**	1	9	-	200	**93**	2	5	-	200	**95**	2	3	-	200	**95**	3	2	-	200
b6	**88**	5	7	1	200	**89**	5	6	-	200	**90**	4	6	-	200	**91**	4	5	-	200
b7	**48**	1	**48**	3	200	**74**	5	21	1	200	**83**	6	11	-	200	**89**	6	5	-	200
b8	**46**	-	**51**	3	200	**57**	4	36	2	200	**74**	3	21	1	200	**81**	4	14	1	200
b9	42	-	**56**	2	200	**66**	9	24	1	200	**73**	8	18	-	200	**83**	7	9	-	200
b10	**47**	-	**50**	3	200	**65**	2	30	2	200	**73**	4	22	1	200	**82**	5	13	1	200
b11	38	-	**61**	1	4	**66**	3	31	-	200	**68**	9	23	-	200	**83**	7	10	-	200
b12	**94**	-	5	1	10	**99**	1	-	-	200	**99**	-	-	-	200	**99**	1	-	-	200
b13	**43**	9	34	14	4	**69**	20	11	-	194	**69**	21	10	-	200	**71**	19	10	-	200
b14	-	-	**78**	22	24	**60**	23	17	-	200	**63**	22	15	-	200	**70**	19	11	-	200
b15	41	**53**	5	-	8	**97**	2	1	-	200	**98**	1	1	-	200	**98**	1	1	-	200
b16	**64**	28	6	-	44	**76**	21	2	-	200	**82**	17	1	-	200	**90**	9	1	-	200
b17	33	**66**	1	-	200	**61**	39	-	-	200	**75**	25	-	-	200	**87**	13	-	-	200

gadget is shared. For example, at $p = 10\%$ the rate of pairs that do not share any gadgets ranges from 63% (*b14*) to 99% (*b12*).

Related approaches (discussed in Section 5) report the *average srate* across all pairs for different benchmark sets. Pappas *et al.*'s zero-cost approach [25] achieves an average *srate* between $74\% - 83\%$ without code degradation, comparable to DivCon's $41\% - 99\%$ at $p = 0\%$. Homescu *et al.*'s statistical approach [15] reports an average *srate* between $82\% - 100\%$ with a code degradation of less than 5%, comparable to DivCon's $83\% - 100\%$ at $p = 5\%$. Both approaches report results on larger code bases that exhibit more opportunities for diversification. We expect that DivCon would achieve higher overall survival rates on these code bases compared to the benchmarks used in this paper.

5 Related Work

There are many approaches to software diversification against cyberattacks. The majority apply randomized transformations at different stages of the software development, while a few exceptions use search-based techniques [20]. This section focuses on quality-aware software diversification approaches.

Superdiversifier [17] is a search-based approach for diversifying software against cyberattacks. Given an initial instruction sequence, the algorithm generates a random combination of the available instructions and performs a verification test to quickly reject non equivalent instruction sequences. For each non-rejected sequence, the algorithm checks semantic equivalence between the original and the generated instruction sequences using a SAT solver. Superdiversifier affects the code execution time and size by controlling the length of the generated sequence. Along the same lines, Lundquist *et al.* [24,23] use program synthesis for generating program variants against cyberattacks, but no results are available yet. In

comparison, DivCon uses a combinatorial compiler backend that measures the code quality using a more accurate cost model that also considers other aspects, such as execution frequencies.

Most diversification approaches use randomized transformations to generate multiple program variants [20]. Unlike DivCon, the majority of these approaches do not control the quality of the generated variants during diversification but rather evaluate it afterwards [10,34,18,14,5,9]. However, there are a few approaches that control the code quality during randomization.

Some compiler-based diversification approaches restrict the set of program transformations to control the quality of the generated code [9,25]. For example, Pappas *et al.* [25] perform software diversification at the binary level and apply three zero-cost transformations: register randomization, instruction schedule randomization, and function shuffling. In contrast, DivCon' combinatorial approach allows it to control the aggressiveness and potential cost of its transformations: a cost overhead limit of 0% forces DivCon to apply only zero-cost transformations; a larger limit allows DivCon to apply more aggressive transformations, potentially leading to higher diversity.

Homescu *et al.* [15] perform only garbage (`nop`) insertion, and use a profile-guided approach to reduce the overhead. To do this, they control the `nop` insertion probability based on the execution frequency of different code sections. In contrast, DivCon's cost model captures different execution frequencies, which allows it to perform more aggressive transformations in non-critical code sections.

6 Conclusion and Future Work

This paper introduces DivCon, a CP approach to compiler-based, quality-aware software diversification against code-reuse attacks. Our experiments show that LNS is a promising technique for a CP-based exploration of the space of diverse program, with a fine-grained control on the trade-off between code quality and diversity. In particular, we show that the set of optimal solutions naturally contains a set of diverse solutions, which increases significantly when relaxing the constraint of optimality. Our experiments demonstrate that the diverse solutions generated by DivCon are effective to mitigate code-reuse attacks.

Future work includes investigating different distance measures to further reduce the gadget survival rate, improving the overall scalability of DivCon in the face of larger programs and larger values of parameter k, and examining the effectiveness of DivCon against an actual code-reuse exploit.

Acknowledgements. We would like to give a special acknowledgment to Christian Schulte, for his critical contribution at the early stages of this work. Although no longer with us, Christian continues to inspire his students and colleagues with his lively character, enthusiasm, deep knowledge and understanding. We would also like to thank Linnea Ingmar and the anonymous reviewers for their useful feedback, and Oscar Eriksson for proof reading.

References

1. Alaba, F.A., Othman, M., Hashem, I.A.T., Alotaibi, F.: Internet of Things security: A survey. Journal of Network and Computer Applications **88**, 10–28 (Jun 2017). https://doi.org/10.1016/j.jnca.2017.04.002, http://www.sciencedirect.com/science/article/pii/S1084804517301455
2. Baudry, B., Monperrus, M.: The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. ACM Comput. Surv. **48**(1), 16:1–16:26 (Sep 2015). https://doi.org/10.1145/2807593, http://doi.acm.org/10.1145/2807593
3. Birman, K.P., Schneider, F.B.: The monoculture risk put into context. IEEE Security & Privacy **7**(1), 14–17 (2009)
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented Programming: A New Class of Code-reuse Attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 30–40. ASIACCS '11, ACM, New York, NY, USA (2011)
5. Braden, K., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., Sadeghi, A.R.: Leakage-Resilient Layout Randomization for Mobile Devices. In: Proceedings 2016 Network and Distributed System Security Symposium. Internet Society, San Diego, CA (2016)
6. Castañeda Lozano, R., Carlsson, M., Blindell, G.H., Schulte, C.: Combinatorial Register Allocation and Instruction Scheduling. ACM Trans. Program. Lang. Syst. **41**(3), 17:1–17:53 (Jul 2019)
7. 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)
8. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne, Australia (2011)
9. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A., Brunthaler, S., Franz, M.: Readactor: Practical Code Randomization Resilient to Memory Disclosure. In: 2015 IEEE Symposium on Security and Privacy. pp. 763–780 (May 2015)
10. Davi, L.V., Dmitrienko, A., Nrnberger, S., Sadeghi, A.R.: Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. pp. 299–310 (2013), tex.organization: ACM
11. Gecode Team: Gecode: Generic constraint development environment (2020), https://www.gecode.org
12. Hamming, R.W.: Error detecting and error correcting codes. The Bell system technical journal **29**(2), 147–160 (1950)
13. Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding Diverse and Similar Solutions in Constraint Programming. In: National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference. p. 6 (2005)
14. Homescu, A., Jackson, T., Crane, S., Brunthaler, S., Larsen, P., Franz, M.: Large-Scale Automated Software Diversity—Program Evolution Redux. IEEE Transactions on Dependable and Secure Computing **14**(2), 158–171 (Mar 2017)
15. 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)

16. Ingmar, L., de la Banda, M.G., Stuckey, P.J., Tack, G.: Modelling diversity of solutions. In: Proceedings of the thirty-fourth AAAI conference on artificial intelligence (2020)
17. Jacob, M., Jakubowski, M.H., Naldurg, P., Saw, C.W.N., Venkatesan, R.: The Superdiversifier: Peephole Individualization for Software Protection. In: Matsuura, K., Fujisaki, E. (eds.) Advances in Information and Computer Security. pp. 100–120. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2008)
18. 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)
19. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131?183 (Jun 1992). https://doi.org/10.1145/130844.130856, `https://doi.org/10.1145/130844.130856`
20. 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)
21. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization. IEEE (2004)
22. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: International Symposium on Microarchitecture. pp. 330–335. IEEE (1997)
23. Lundquist, G.R., Bhatt, U., Hamlen, K.W.: Relational processing for fun and diversity. In: Proceedings of the 2019 miniKanren and relational programming workshop. p. 100 (2019)
24. Lundquist, G.R., Mohan, V., Hamlen, K.W.: Searching for Software Diversity: Attaining Artificial Diversity Through Program Synthesis. In: Proceedings of the 2016 New Security Paradigms Workshop. pp. 80–91. NSPW '16, ACM, New York, NY, USA (2016)
25. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In: 2012 IEEE Symposium on Security and Privacy. pp. 601–615 (May 2012), iSSN: 1081-6011
26. Petit, T., Trapp, A.C.: Finding Diverse Solutions of High Quality to Constraint Optimization Problems. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (Jun 2015)
27. Salwan, J.: ROPgadget Tool (2020), `http://shell-storm.org/project/ROPgadget/`
28. 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)
29. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 1520, pp. 417–431. Springer (1998)
30. SPEC: CPU 2006 Benchmarks (2020), `https://www.spec.org/cpu2006`, accessed on 2020-03-20
31. Sweetman, D.: See MIPS Run, Second Edition. Morgan Kaufmann (2006)
32. Van Hentenryck, P., Coffrin, C., Gutkovich, B.: Constraint-Based Local Search for the Automatic Generation of Architectural Tests. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming - CP 2009. pp. 787–801. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2009)

33. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM (JACM) **21**(1), 168–173 (1974)
34. Wang, S., Wang, P., Wu, D.: Composite Software Diversification. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 284–294 (Sep 2017)