Faultprog: Testing the Accuracy of Binary-Level Software Fault Injection

Domenico Cotroneo, Anna Lanzaro, and Roberto Natella

Abstract—Off-The-Shelf (OTS) software components are the cornerstone of modern systems, including safety-critical ones. However, the dependability of OTS components is uncertain due to the lack of source code, design artifacts and test cases, since only their *binary code* is supplied. *Fault injection* in components' binary code is a solution to understand the risks posed by buggy OTS components.

In this paper, we consider the problem of the *accurate mutation of binary code* for fault injection purposes. Fault injection emulates bugs in high-level programming constructs (assignments, expressions, function calls, ...) by mutating their translation in binary code. However, the semantic gap between the source code and its binary translation often leads to inaccurate mutations.

We propose *Faultprog*, a systematic approach for testing the accuracy of binary mutation tools. Faultprog automatically generates synthetic programs using a stochastic grammar, and mutates both their binary code with the tool under test, and their source code as reference for comparisons. Moreover, we present a case study on a commercial binary mutation tool, where Faultprog was adopted to identify code patterns and compiler optimizations that affect its mutation accuracy.

Index Terms—Off-The-Shelf software; Dependability Benchmarking; Fault Injection; Software Mutation; Random Testing.

1 Introduction

The need for lowering development costs and the time-to-market leads companies to develop their systems by integrating Off-The-Shelf (OTS) software components from third parties. Nowadays, this approach is also adopted for safety-critical systems such as avionic, automotive and medical ones, where the dependability of OTS software becomes a major concern [1], [2], [3]. When an OTS component is reused in a new context, the system may use parts of the component that were not previously used and/or that were only lightly tested. Furthermore, the OTS component may interact with the new environment in unforeseen ways, thus exposing *residual software faults* ("bugs") that had not been discovered in previous use. The issue of OTS software is exacerbated by the lack of source code, design artifacts and test cases, since only its executable *binary code* is usually available.

Fault injection is a widely-used approach for mitigating the risks posed by faulty OTS components in a critical system [4], [5], [6]. It consists in the deliberate injection of faults into a component, by means of **binary mutation** (BM). Mutations mimic software faults by inserting small "faulty" changes into the binary code of the OTS component; then, the system is executed, in order to analyze its ability to tolerate faults in the OTS component, as demonstrated in a variety of work [7], [8], [9], [10]. Another emerging application of BM is to increase the efficiency of mutation testing (which uses mutants to select test cases), by avoiding the cost of recompiling a high number of source-code mutants [11], [12].

However, making accurate changes to binary code is a technically-challenging problem [13], [14], [15], [16], [17]. A BM tool works in two steps: (i) it first looks for *code patterns* in the binary code that represent high-level programming constructs, according to a *fault model* to inject [18], [19], [14]; (ii) the code pattern is mutated by replacing its machine instructions. Fig. 1 shows an example: To emulate a "missing variable assignment" fault (a frequent type of software fault, as discussed later), a BM tool should identify, and remove, the machine instructions that result from the translation of an assignment in the source code. The removal of these instructions from the binary code emulates a programmer's omission in the source code.

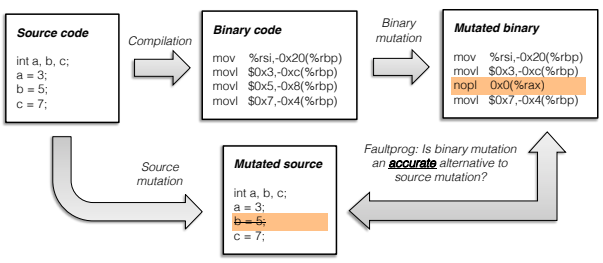


Fig. 1. Example of fault injection at binary level.

The main challenge for a BM tool is to assure the **accuracy** of binary mutations, that is: binary mutations should emulate the software faults as they would be injected into source code and then compiled into binary code [13], [14], [17]. Accuracy is a necessary

• *Authors are with the Department of Electrical Engineering and Information Technologies (DIETI), Federico II University of Naples, Italy. E-mail: {cotroneo, anna.lanzaro, roberto.natella}@unina.it*

condition for the **rigorous and unbiased** evaluation of fault tolerance properties [8], [7], [9], [10], since injecting inaccurate mutants would lead to misleading conclusions (which can be very dangerous in the context of critical systems). However, the accuracy of binary mutation is hampered by the semantic gap between source code (consisting of high-level programming constructs wrote by developers) and its translation into binary code (consisting of low-level machine instructions). The gap is widened by the complexity of programming languages, compilers, and CPUs, and makes difficult to identify high-level programming constructs by only looking at their translation in binary code. This gap negatively affects the accuracy of BM tools, as showed in previous work [15] (see § 2 for a more detailed discussion), and leads to an incorrect assessment of fault tolerance.

In this work, we propose an approach, namely *Faultprog*, for **testing the accuracy of BM tools**. Faultprog is based on the automatic generation of *synthetic programs*, which are submitted as inputs to a BM tool, in order to evaluate its accuracy at performing binary mutations. First, several synthetic programs are generated by encompassing different programming constructs (e.g., expressions, function calls, assignments) in different contexts (e.g., nested loops, control flow constructs and function calls). Then, the BM tool is applied on the binary code of these synthetic programs. The binary mutants produced by the BM tool are compared against the mutants produced by source-code mutation and compilation. In this process, the analysis of source code serves as a reference, as it does not suffer the limitations of binary-level mutation. Faultprog assesses the **accuracy of the BM tool at correctly recognizing and mutating programming constructs at the binary level**, revealing its issues and limitations. In other words, the synthetic programs act as a "test suite" for assessing the quality of binary mutations. Given the increasing use of fault injection techniques in industry, which are nowadays recommended by several safety standards [20], we believe that giving an approach to practitioners to evaluate the accuracy of fault injection tools is very important.

This paper evaluates the feasibility and the usefulness of putting the Faultprog approach into practice in the context of a BM tool from a commercial fault injection suite. We use Faultprog to test a tool currently under development by CRITICAL Software S.A., a leading company in the field of independent V&V of safety-critical systems. We focus on the in-depth analysis of this BM tool, and present the lessons learned and the generality and limitations of the findings. Overall, the use of random program generation in Faultprog proved to be useful at exposing the BM tool to complex program expressions, and to uncover corner cases that cause inaccurate mutations. While the BM tool produced correct mutations in many cases, we found that some code patterns in the fault model ("constraints") are not correctly handled, and that additional code patterns are needed to identify all injectable faults. Finally, we found that only few compiler optimizations have a significant impact on the accuracy of the BM tool, and developers' efforts should be focused on improving the tool with respect to these optimizations.

The paper is structured as follows: § 2 discusses the problem of BM accuracy with a motivating example; § 3 describes Faultprog; § 4 presents the experimental results obtained in the context of a BM tool; § 5 discusses related work; § 6 closes the paper.

2 The problem of accuracy

As a motivating example for the rest of the paper, we summarize the findings of our preliminary work on the accuracy of BM [15]. In that work, we analyzed a BM tool that adopts the *Generic Software Fault Injection Technique* (G-SWFIT) [14], a well-known technique for injecting realistic software faults in binary code. The peculiarity of G-SWFIT is that its fault model reflects *the types of software faults that most frequently cause failures*, according to field data about failures experienced by users [18], [19], [14]. The definition of these fault types is based on the well-known *Orthogonal Defect Classification* [21], which classifies software faults as *Assignment*, *Algorithm*, *Checking*, and *Interface* faults.

For each fault type (listed in TABLE 1), G-SWFIT defines: a code pattern where the fault type should be injected (for instance, assignment faults should be injected in *move* instructions, and control flow faults should be injected in *branch* instructions), and a code change to be injected for emulating that fault type (e.g., replacing *move* or *branch* instructions with *nop*s). Code patterns reflect the typical translation of high-level programming constructs of the fault model into sequences of machine instructions. The definition of code patterns is the most tricky aspect of G-SWFIT, since the binary translation of programming constructs is influenced by several factors, including the programming language, the compiler and its optimizations, and the hardware architecture.

In addition to the code patterns, G-SWFIT's fault types also include a set of rules (*constraints*) that define the "context" in which faults can be injected. The constraints serve to avoid code locations where the change would not emulate a realistic fault [14]. These constraints, listed in TABLE 1 and in TABLE 2, are based on the analysis of field data (see [22] for details). Constraints may apply to more than one fault type.

For example, the MFC fault type has a constraint (C01) imposing that a function call should be removed only if it does not return any value or the return value is discarded by the caller. In fact, field data (and also intuition) point out that if a programmer uses the return value from a function call, then it is unlikely to forget that function call in the program, and removing

TABLE 1
Fault types of G-SWFIT [14], [22].

Class	Acronym	Description	Constraints
Assignment	MVIV	Missing variable initialization using a value	C02, C03, C04, C05, C06
	MVAV	Missing variable assignment using a value	C02, C03, C06, C07
	MVAE	Missing variable assignment with an expression	C02, C03, C06, C07
	WVAV	Wrong value assigned to variable	C03, C06, C07
Algorithm	MFC	Missing function call	C01, C02
	MIFS	Missing IF construct + statements	C02, C08, C09
	MIEB	Missing IF construct + statements + ELSE construct	C08, C09
	MLPA	Missing small and localized part of the algorithm	C02, C10
Checking	MIA	Missing IF construct around statements	C08, C09
	MLAC	Missing AND in expression used as branch condition	
	MLOC	Missing OR in expression used as branch condition	
Interface	WPFV	Wrong variable used in parameter of function call	C03, C11
	WAEP	Wrong arithmetic expression in function call parameter	

TABLE 2
Constraints of fault types in G-SWFIT [14], [22].

Id	Description
C01	Return value of the function must not being used
C02	The construct must not be the only statement in the block
C03	Variable must be local
C04	Must be the first assignment for that variable in the module
C05	Assignment must not be inside a loop
C06	Assignment must not be part of a FOR construct
C07	Must not be the first assignment for that variable in the module
C08	The IF construct must not be associated to an ELSE construct
C09	The block must not include more than five statements and not include loops
C10	Statements are in the same block, do not include more than 5 statements nor loops
C11	There must be at least two variables in this module

the function call would not be representative of real software faults made by programmers. Another example is the MLPA fault type, which has a constraint (C10) that imposes to remove at least two, and at most five, consecutive statements that are neither control nor loop statements. All these constraints are important for emulating software faults in a representative way, which in turn is a requirement for a correct and unbiased evaluation of fault tolerance [23].

In our previous experiment [15], we evaluated the accuracy of a G-SWFIT fault injection tool on a complex software system for the space domain. In that experiment, we injected faults in both OS and application binary code, and compared binary mutations with mutations performed on the source code, following the rules of G-SWFIT in both cases. We found that: (i) for each fault injected by the BM tool, there were two more injectable faults that were omitted by the tool, and (ii) among the faults injected by the BM tool, about half of them were invalid. These incorrect injections were due to two reasons: (i) intrinsic limitations of G-SWFIT, and of BM in general, that are very difficult to avoid (e.g., *inline* C functions, which mislead a BM tool to inject one separate fault for each call site of the function); (ii) incorrect injections due to the incomplete or simplified implementation of the BM tool. Many incorrect injections were due to the second cause, and in particular were related to the implementation of fault constraints and to the identification of code blocks and control structures. For instance, spurious faults were in some cases incorrectly injected when the target instruction was the only statement within a block, and some faults were omitted when an *if* construct included a *return* statement. These incorrect injections were not due to intrinsic problems of BM, and could be avoided by conducting a rigorous testing of the BM tool.

These results motivated us to develop a systematic approach for testing BM tools. The experimental methodology of our previous study [15] cannot be easily adopted by developers of BM tools, since it was based on the injection of a large number of faults (tens of thousands in our experiment), but analyzing even a sample of these faults required considerable efforts. Moreover, the analysis is focused on a specific software under study, and incurs a significant number of incorrect injections not due to issues in the BM tool (which are of interest for developers of BM tools), but to intrinsic or known limitations of BM. Therefore, in this paper we propose a novel approach that evaluates injection accuracy on synthetic programs, which are generated in a controlled and automated way in order to make the analysis more efficient.

3 The Faultprog approach

Faultprog automates the evaluation of BM tools at accurately injecting faults in binary code. The approach uses *synthetic programs*, i.e., programs (in a high-level language, such as C) that are automatically and randomly generated with the sole purpose to evaluate the ability of the BM tool to inject faults

into them. These synthetic programs are compiled into binary code, and fed to the BM tool. Then, we analyze the mutations obtained from the BM tool.

The idea of this approach is to generate several synthetic programs, in such a way to expose the BM tool to code patterns that could point out its inaccuracies. Synthetic programs (Fig. 2) contain a *target fault location* in their code, where the BM tool is expected to inject a fault. The target fault location is generated to comply with the fault types and constraints for which the BM tool is designed. For instance, to evaluate the "missing assignment using value" (MVAV) fault type of G-SWFIT (Table 1), we generate a target fault location containing an assignment statement. To comply with the constraints of fault types of G-SWFIT (Table 2), the target fault location consists of an assignment made to a *local* variable (constraint C03), and this assignment is not the only instruction of its block (constraint C02). If the tool is not able to inject a fault in the target fault location, then the synthetic program exposes an issue of the fault injector, i.e., it exhibits an **omitted injection**.

Moreover, we also generate synthetic programs in which the fault constraints are deliberately *not satisfied*, and in which the fault injector should avoid to inject faults. If the fault injector fails to recognize that the target fault location is not compliant to the fault model, it will erroneously inject a fault in the target fault location. In such a case, the synthetic program exposes an issue of the BM tool. This situation represents a **spurious injection**.

Finally, it is possible that the BM tool correctly identifies the target fault location, but it does not inject the intended fault. For instance, the BM tool may mutate both the target fault location, and other statements that surround the target but that should not be mutated. In such cases, executing the mutated program will produce different results than the same program mutated at the source code level. This situation represents an **incorrect injection**.

To complete the synthetic programs, and to evaluate the accuracy of the BM tool in the presence of complex code patterns, the target fault location is surrounded, preceded and followed by additional randomly-generated programming constructs, that represent respectively the *context*, the *preamble* and the *postamble* of the target fault location (Fig. 2).

Fig. 3 summarizes the workflow and the tools involved in the Faultprog approach. A *program generator* produces synthetic programs based on the structure of Fig. 2, by using a *grammar* of the source language to generate syntactically-valid statements, and a *fault model* to generate different flavors of the target fault location. The program is fed both to the BM tool that we are testing (**Tool Under Test**) and to a source mutation tool (**Oracle Tool**) that injects faults in source code, rather than binary. The Oracle Tool serves as a reference for evaluating the accuracy of

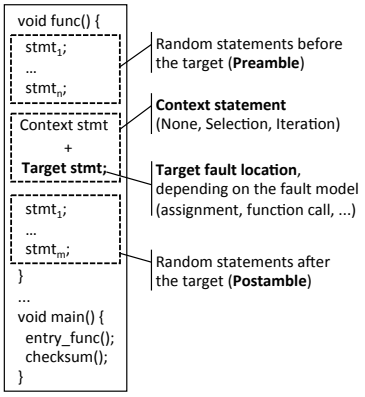


Fig. 2. General structure of a synthetic program.

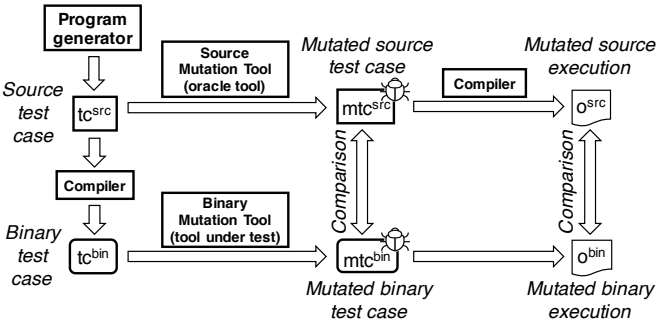


Fig. 3. The Faultprog approach.

the Tool Under Test: it follows the same fault model (e.g., G-SWFIT), but works on source code, thus it avoids the accuracy issues encountered by the BM tool. The mutated programs are statically and dynamically compared. Program generation, mutation and comparison are automatically repeated, by leveraging a compiler and a build system.

Test-suite generation (§ 3.1): We automatically generate a set of synthetic programs, which are **test-cases** for the BM tool, and form a **test-suite**. Programs are generated according to the *fault types* and *constraints* that we are testing, where the *target* statement is surrounded by a *context*, i.e., iteration (a loop construct) or selection (a conditional construct) statements. The test-suite is obtained by generating different combinations of these parameters, while also controlling the nesting level and type of expressions (e.g., constants, variables, arithmetic operations). The test-suite ($TS^{\mathtt{src}}$) contains both **valid** test-cases (i.e., test-cases that satisfy all the constraints for a fault type, and where the tool is expected to inject), and **invalid** test-cases (i.e., test-cases that do not satisfy one of the constraints, and that the tool should not mutate). Test-cases are compiled into binary code ($TS^{\mathtt{bin}}$).

Test-suite mutation (§ 3.2): The Tool Under Test takes binary test cases from $TS^{\mathtt{bin}}$ as input, and produces faulty versions of these programs (*mutated test suite*, $MTS^{\mathtt{bin}}$) by changing their binary code. At the same time, the test-suite $TS^{\mathtt{src}}$ in source-code form is

fed to the Oracle Tool. For each test-case $tc^{\texttt{bin}}$ in $TS^{\texttt{bin}}$, we collect the mutants $MTS^{\texttt{bin}}$ generated by the Tool Under Test, and the output $o^{\texttt{bin}}$ resulting from the execution of $mtc^{\texttt{bin}}$. The same process is performed on the source-code test-suite $TS^{\texttt{src}}$ using the Oracle Tool, producing the mutants $tc^{\texttt{src}}$ and the outputs $o^{\texttt{src}}$.

Comparison of mutants (§ 3.3): The mutants produced by the tools ($mtc_i^{\texttt{src}}$ and $mtc_i^{\texttt{bin}}$) are compared, in terms of mutated instructions, and outputs from their execution ($o_i^{\texttt{src}}$ and $o_i^{\texttt{bin}}$). The comparisons determine whether the BM tool did *spurious* (i.e., mutations injected in the binary code, but not in the source code), *omitted* (i.e., mutations injected in the source code, but not in the binary code) or *incorrect* injections (i.e., mutations that lead to different outputs).

3.1 Test-suite generation

The proposed approach is based on the random generation of synthetic programs. We extend the use of random programs, that were adopted in past studies for testing compilers, interpreters, and program analyzers [24], [25], [26], [27], to test the accuracy of binary mutation tools. Random program generators produce programs as a sequence of statements, including global and local variables declarations, functions, assignment, expressions, selection and iteration statements. The inputs of these programs are constants produced during the random generation process. The output of these programs is a checksum of the global variables of the program, which is computed just before the termination of the program.

A *synthetic program* is a sequence of randomly generated statements. A statement can be an expression, an assignment, a function call, a selection or an iteration statement. The random generator bases the program generation on a *stochastic grammar* of the language [28]. A stochastic grammar associates probabilities to each grammar rule of the language. Each rule consists of a *left side* and a *right side*, where the left side is a non-terminal symbol, and the right side contains one or more sequences of symbols (either terminal and non-terminal). A statement is generated by concatenating *terminal* (e.g., operators like "+" and "->", or keywords like "for") and *non-terminal symbols* in the sequence. Beginning from a "start" rule, the program generator replaces each non-terminal symbol, by recursively applying the rules in the grammar, until there are no more non-terminal symbols. When the right side contains more than one sequence, the stochastic grammar associates a probability to each sequence, and the program generator randomly selects a sequence on the basis of its probability.

In the Faultprog approach, we modify this random program generation process to follow the structure showed in Fig. 2. The random program should contain a statement, named *target*, that is the location for injecting faults according to the fault model. The target statement is generated randomly, according to the following parameters (TABLE 3):

- *Fault type* that has to be tested. For instance, when testing the MVAE fault type of G-SWFIT, the target fault location consists of a local variable assignment with an expression, such as an arithmetic expression.
- *Fault constraint* to be violated (if any). For instance, when testing the MFC fault type in G-SWFIT, we can generate *valid* programs that comply to both constraints *C01* and *C02*, and *invalid* programs that violate one of these two constraints (e.g., the target statement is a function call whose return value is used in the rest of the program).
- *Context* in which the target statement has to be inserted. It can be a *selection* (e.g., if-then-else construct) or an *iteration* statement (e.g., while- or for-loop construct).
- *Nesting* depth of the context, such as the number of nested loops in which the target statement should be contained.
- *Type of basic operand* (BO) to use in expressions of the target statement. They can be constants, global or local variables, or function calls.
- *Structure of expressions* in the target statement. According to this parameter, the target statement contains expressions that are obtained by different combinations of one or more BOs, random sub-expressions and random operators.

To obtain test-suites, we generate several random programs using the *Faultprog* automated program generator. We generate test cases to cover all the fault types in the fault model. Moreover, we consider all combinations of parameters of Table 3 that apply for each fault type. For instance, given the MFC fault type, we generate three sets of test cases: (i) both constraints *C01* and *C02* are satistified; (ii) *C01* is not satisfied; (iii) *C02* is not satisfied.

Fig. 4 shows an example of a random (valid) program, in which (i) the MFC fault type is selected, (ii) all constraints are satisfied, (iii) a function call is nested in two loops, and (iv) the expression in the target statement (in this case, the parameter of the function call) is the sum of two local variables.

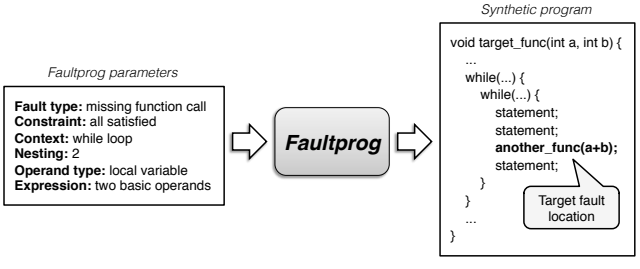


Fig. 4. An example of synthetic program for testing the MFC ("Missing Function Call") fault type.

In this study, we tailored the *Faultprog* random

TABLE 3
Parameters of the *Faultprog* program generator.

Parameter	Description	Options
Fault type	Type of target statement, according to the fault model.	• MFC • MIFS • MIEB ...
Fault constraint	Constraint to be violated (if any).	• all satisfied • C01 not satisfied ...
Context	The statements surrounding the target statement.	• none • *while* loop • *for* loop • *if*-target • *if*-target-*then-else* • *if*-*then-else*-target
Nesting	The nesting depth of the context statements.	• 0 • 1 • 2
Basic operand (*BO*) type	The type of operands involved in the target statement.	• constant • local variable • global variable • function call
Expression structure	The expression involved in the target statement.[1]	• single basic operand (BO) • expression with two BOs • expression with three BOs • expression with a BO and a random sub-expression • expression with a BO and two random sub-expressions

[1] Expressions are obtained by a random combination of one or more basic operands, sub-expressions and operators.

program generator for the C language, since this language is predominant in safety-critical control systems and systems software. However, the general approach behind *Faultprog* can easily be ported to other programming languages, such as C++, by including additional programming constructs in the program generation (e.g., by including pointers as basic operands in the target statement). In fact, this approach has been adopted for many languages, such as SQL [29], Java [30], and x86 machine code [31]. Sirer and Bershad [30] showed how grammars can be customized using a domain specific language. We implemented *Faultprog* by enhancing the open-source *Randprog* tool (that was originally aimed for testing C compilers [32]), to support the evaluation of BM tools.

3.2 Test-suite mutation

After generating test-suites (both in source-code form, $TS^{\mathtt{src}}$, and in binary-code form, $TS^{\mathtt{bin}}$), we apply the Oracle Tool and the Tool Under Test on them. For each synthetic program, we store mutants produced by the tools, i.e., the sets $MTS^{\mathtt{src}}$ and $MTS^{\mathtt{bin}}$.

For each binary test case $tc^{\mathtt{bin}}$, we focus the analysis on mutations injected in the binary instructions that correspond to the *target fault location* of the synthetic program $tc^{\mathtt{src}}$. Mutations in other parts the program that do not belong to the target fault location, such as the preamble, the postamble and the context (Fig. 2), are not considered in the analysis, since they are only meant to introduce complex code patterns around the target fault location, and are aimed at evaluating the ability of the BM tool at recognizing the target fault location among other binary instructions.

The location of target statement in the source code is recorded by the program generator (Fig. 3) when a synthetic program $tc^{\mathtt{src}}$ is created. The synthetic program is then compiled into a binary program $tc^{\mathtt{bin}}$. To identify the binary instructions in $tc^{\mathtt{bin}}$ that correspond to the target fault location, we leverage *debugging information* in the binary program, that is, auxiliary information inserted by the compiler in the program to track the correspondence between program elements (such as statements) and their binary translation [33]. Even if this information is not available in OTS software components, we still insert it when compiling the synthetic program, in order to provide a "ground truth" for evaluating the BM tool (as discussed in § 3.3). We remark that debugging information does not interfere with the binary code of synthetic programs: compilers insert debugging information in a distinct section of an executable file, which is separated from the binary code. Therefore, the binary instructions are the same that would be generated when debugging information is disabled.

After applying the Oracle Tool and the Tool Under Test, we obtain for each synthetic program one or more mutants of the source code, and one or more mutants of its binary code. It is possible that more than one mutant is generated when the target fault location is a complex statement, and more than one fault could be potentially injected in that location (e.g., the "missing arithmetic expression in function parameter", when the function call contains several parameters). These synthetic programs are handled as a set of several distinct test cases.

3.3 Comparison of mutants

The mutants from both the Oracle Tool and the Tool Under Test are compared to identify *omitted*, *spurious* and *incorrect* injections by the Tool Under Test. Ideally, these tools should inject the same mutation at the binary- and at the source-level. First, we check whether the binary instructions mutated by the Tool Under Test are the ones corresponding to the target fault location (denoted by debugging information, § 3.2). Then, we run the mutants generated by both tools, and compare their behavior. The Tool Under Test injects correctly in two cases:

1) In the case of *valid* synthetic programs (i.e., the target statement satisfies all the constraints imposed by the fault type being tested): the BM tool identifies and mutates all the binary instructions of the target fault location, and the

resulting mutant produces the same outputs of the corresponding source mutant.
2) In the case of *invalid* synthetic programs (i.e., the target statement does not satisfy one of the constraints imposed by the fault type being tested): both tools identify the target statement as an invalid location where *not* to inject a fault, and thus do not produce mutants.

On the contrary, the Tool Under Test fails when:

1) In the case of *valid* synthetic programs: the Oracle Tool mutates the target fault location, while the Tool Under Test does not mutate the corresponding binary instructions (denoted by the debugging information). The test case detects an **omission** of the Tool Under Test.
2) In the case of *invalid* synthetic programs: the Oracle Tool *does not* mutate the target fault location, but the Tool Under Test mutates the binary instructions of the target location (denoted by the debugging information). The test case detects a **spurious** injection of the Tool Under Test.
3) In the case of *valid* synthetic programs: the BM tool injects a fault in the target binary instructions, but the execution results differs from the mutant by the Oracle Tool. The test case detects an **incorrect** injection of the Tool Under Test.

In order to achieve sufficient confidence that a synthetic program is accurately mutated, we execute it under several random inputs. The inputs exercise the mutated statement, and lead it to "infect" the state of the program (in the sense of the PIE model by Voas [34]). This approach (and any other approach) cannot *prove* that two mutants behave in the same way for all possible inputs, since this problem is *undecidable*: it is the same problem of detecting *equivalent mutants* in mutation testing [35], [36], [37]. However, as discussed in recent work on mutation testing [38], random inputs can estimate the amount of equivalent mutants, with a high degree of statistical confidence.

Given that the two tools should inject the same faults, we expect that binary-level and source-level injections have the same effects on the target program, in terms of impact on the control flow of the execution and on the state of the program. We detect differences in the two executions by computing a *checksum* of global variables, and by comparing these two checksums. We deliberately generate programs that make frequent use of global variables and of the variables in the target statement, in order to let the effects of the fault to surface quickly. For example, assuming that the target statement is an *if* construct, and that the fault removes its *else* branch or changes the logical condition of the *if*, then the program state will be immediately influenced by the lack of the accesses to global variables made within the *else* branch. It is important to note that our objectives do not require us to fully cover the code of the randomly-generated programs, since they are not intended to make any meaningful computation, and thus we are not interested in testing their correctness.

When we run the program with random inputs, we check whether the target statement is covered. If the target statement is not covered because of an infinite loop, or because of a control-flow construct that contains the target statement, we make a small change in the *preamble* and *context* of the synthetic program, in order to steer the execution towards the target statement. The change retains the features of the Faultprog test: we keep unchanged the target statement, and the program structure around the target statement; we only make minimal changes to expressions in the control condition of loop and control-flow constructs, to guide the execution on the desired path.

First, we enforce a maximum number of iterations for loops that do not terminate in a reasonable amount of time. This guarantees that the program does not get stuck in infinite loops, and that function calls return in a limited amount of time. Second, we change the conditional expression of control-flow constructs that are in the path of the target statement, but that hinder its execution. When the target statement is not covered, we identify the covered control-flow construct closest to the target statement. Then, its control condition is negated both in the source mutant and in the binary mutant, and execute them again. These steps are repeated for each conditional expression in the path before the target statement, until the target statement is covered. This approach guarantees convergence in a limited number of iterations: in the worst case, we need to iterate for all the conditional expressions within the function that contains the target statement.

After performing the comparisons of the mutants, we analyze the results to identify the causes of inaccuracies in the BM tool. To give feedback to developers, we analyze the *distributions* of failed test-cases with respect to Faultprog's parameters, to identify which parameter leads to the highest number of failures. When pinpointing inaccuracies, it is useful to know which fault types, constraints, or contexts caused a significant number of omitted, spurious or incorrect injections. This information enables developers to diagnose problems, by looking at specific areas of the BM tool. Moreover, after fixing the tool, developers can apply again the test-cases to validate the fix (i.e., whether it reduces inaccurate injections).

4 CASE STUDY

We applied the Faultprog approach to a BM tool based on G-SWFIT (Tool Under Test) from the Xception suite. As reference, we use the SAFE tool for source code mutation (Oracle Tool).

4.1 The Xception fault injection tool

Xception is a fault injection tool suite, developed by CRITICAL Software S.A. (under the brand *csXcep-*

tion), for supporting V&V of safety- and mission-critical systems. Xception was originally developed as a *Software-Implemented Fault Injection* (SWIFI) tool, jointly with the University of Coimbra in Portugal. The original Xception, and SWIFI in general, injected hardware faults (such as electromagnetic interferences, and circuit wear-out) by emulating the effects of such faults in software, through random *bit-flipping* of memory and CPU registers. In particular, Xception was designed to exploit the debugging and performance monitoring features of modern CPUs, in order to inject faults with minimal intrusiveness [39].

Xception is today a professional environment for performing fault injection tests. It adopts a modular architecture to support several forms of fault injection including, but not limited to, SWIFI. In particular, Xception is currently being extended with a plug-in to support the G-SWFIT fault injection technique (previously discussed in § 2), for the injection of software faults through binary mutation.

The Experiment Management Environment (EME) is the core of Xception, and is responsible for controlling, monitoring, and storing the results of the experiments. The EME interacts with a plug-in that actually injects faults into the target system. In turn, the G-SWFIT plug-in (Fig. 5a) performs binary mutations on behalf of Xception. The G-SWFIT plug-in disassembles the binary code of the target program, and looks for code patterns (i.e., sequences of machine instructions) that match to the target programming constructs (function calls, assignments, ...), according to the fault types and constraints presented in Section 2. Once code patterns are recognized, they are mutated by replacing machine instructions with *nop* instructions (to emulate a statement omission), or by modifying opcodes and the operands (to emulate an incorrect statement). This plug-in currently supports the *PowerPC hardware architecture* and the *GCC compiler toolchain*, on which we focus our analysis.

4.2 The SAFE fault injection tool

To point out inaccuracies of binary mutation, we also inject source-code mutations in the synthetic programs, which serve as a reference for binary mutations. We adopt the SAFE fault injection tool [23], which implements the same fault types of the original G-SWFIT, but injects these faults in the source code instead of binary code (Fig. 5b). A source code file is first processed by a C/C++ front-end, which builds an Abstract Syntax Tree representation. It then identifies locations where a fault type can be introduced in a syntactically-correct manner, and that comply with fault constraints (§ 2). The source code is then mutated, compiled, and compared to binary mutations as previously discussed. The SAFE tool has extensively been used in several fault injection studies and industrial projects [23], [20], [40], [41], [42], including our

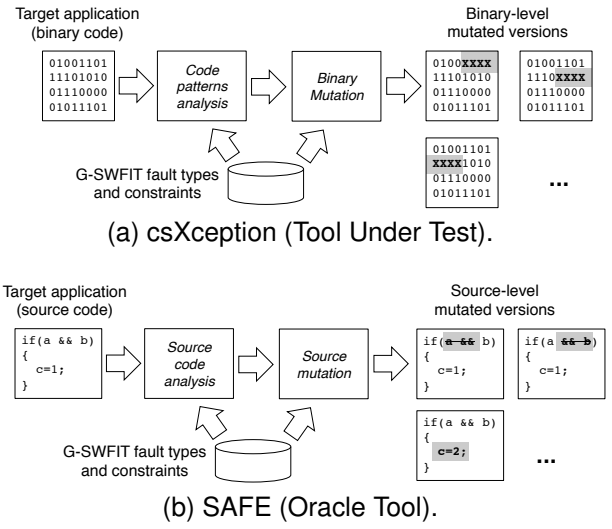


Fig. 5. Fault Injection Tools analyzed in this study.

preliminary study on BM accuracy in [15], and it has been applied on several software systems (including the Linux kernel, the RTEMS real-time OS, and several open-source projects). Based on this experience, the quality of source code mutations produced by SAFE has steadily improved over the years, and it has matured to be a reliable implementation of G-SWFIT.

4.3 Test suite for Xception G-SWFIT

We use Faultprog to generate a test-suite for evaluating the Xception G-SWFIT plug-in. For each fault type, Faultprog generates a set of valid programs (i.e., programs containing a valid target fault location), and a set of invalid programs (i.e., programs containing a target fault location that violates one of the constraints of the fault type, according to TABLEs 1 and 2).

Several programs are generated for each fault type. Test cases cover all possible combinations in the Faultprog parameter space (see § 3.1 and TABLE 3). These parameters determine the content of a random program, by influencing which type of expression should be generated in the target fault location, and which statements should surround it. The number of generated random programs varies across ODC classes, since they depend on the number and on the type of constraints, and on the number of meaningful combinations of parameters for each fault type. We only exclude combinations with conflicts between parameters. For example, the MVAV fault type ("missing variable assignment using a value") imposes that the parameter "expression structure" must be a single operand, and that the "basic operand type" can only be a constant. The number of resulting combinations is showed in TABLE 4. For each combination, we generated 5 random programs for each combination, which allowed to achieve a reasonable code coverage of the BM tool, and to point out important issues.

TABLE 4
Test-suites generated by *Faultprog*

ODC Class	# of valid programs	# of invalid programs	statement coverage (%)
Assignment	416	365	35%
Algorithm	960	1,745	34%
Checking	624	480	41%
Interface	224	42	29%
Total	**2,224**	**2,632**	**72%**

4.4 Experimental measures

In our experiments, we evaluate the accuracy of BM by measuring the percentage of test cases that lead to *spurious* and *omitted* injections, i.e., test cases in which the BM tool injects an incorrect number of faults (§ 3.3). More precisely, the *percentage of omitted injections* of the BM tool is defined as:

$$\text{omitted}_{\%} = \frac{\text{\# source-level injections not matching binary-level injections}}{\text{\# source-level injections}} \quad (1)$$

which is the ratio between (on the denominator) the number of all source-injected test cases (i.e., the injections that the BM tool is ideally expected to inject at the binary-level), and (on the numerator) which have been mutated as source code by the Oracle Tool, but were not mutated as binary code by the Tool Under Test (i.e., *omitted injections*). In a similar way, the *percentage of spurious injections* of the BM tool is:

$$\text{spurious}_{\%} = \frac{\text{\# binary-level injections not matching source-level injections}}{\text{\# binary-level injections}} \quad (2)$$

The percentage is computed from the ratio between (on the denominator) the number of all binary-injected test cases (including both correct and spurious injections), and (on the numerator) the number of test cases which have been mutated as binary code by the Tool Under Test, but were not mutated as source code by the Oracle Tool (i.e., *spurious injections*).

Finally, the percentage of incorrect injections is given by the ratio between (on the denominator) the number of test cases that were mutated both at binary- and at source-level, and (on the numerator) the number of such test cases whose binary and source mutants produced different outputs:

$$\text{incorrect}_{\%} = \frac{\text{\# non-omitted, non-spurious injections where source and binary mutations give different outputs}}{\text{\# non-omitted, non-spurious injections}} \quad (3)$$

4.5 Experimental results

We here analyze omitted, spurious, and incorrect injections (§ 4.4) for the Xception BM tool. First, we consider the most typical scenario, where the Faultprog test-suite is compiled under the *default compiler configuration*. We then evaluate whether *compiler optimizations* impact on the accuracy of the BM tool, by compiling the test-suite with optimizations enabled, and applying it again on the BM tool. The experimental setup included an Intel x86-64 PC running Ubuntu Linux 14.04, the Faultprog program generator, the fault injection tools, and GCC version 4.1.1, which we configured to cross-compile for the *powerpc-eabi* target. We executed the mutated programs on an Apple iBook G4 running Debian Linux 6.0 for PowerPC.

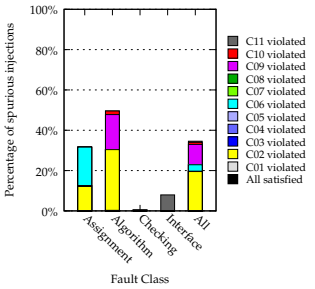
4.5.1 Spurious injections

According to eq. 2, we evaluated the percentage of test cases, grouped by ODC class, in which the BM tool produced spurious faults, showed in the bar plots of Fig. 6. The bar plots show the same percentages, but the bars are splitted with respect to different perspectives (TABLE 3): the percentages are divided respectively by (i) constraint violated by the test case (Fig. 6a), (ii) context of the target fault location (Fig. 6b), (iii) level of nesting (Fig. 6c), (iv) type of operand (Fig. 6d), and (v) structure of the expression in the target fault location (Fig. 6e).

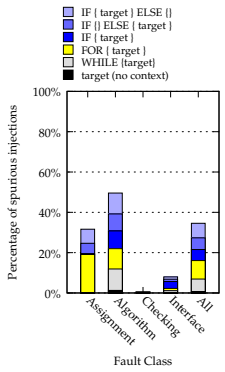
The highest percentage (49.58%) occurred for the *Algorithm* class, and in particular the MIEB, MFC, and MLPA fault types. In Fig. 6a, constraints C02 and C09 account for most of the spurious injections (respectively, 19.37% and 10.05% of all test cases). This means that there was a high number of incorrect injections even if the target statement was the only statement in its block (but a fault should *not* be injected, since the test case violates constraint C02) or the *if* construct included more than five statements (a fault should *not* be injected, since constraint C09 is violated).

An example of spurious injection for the MFC fault type is showed in Fig. 7. The function call *func_7()* is the only instruction in a *while* block, and it is not a valid location according to constraint C02: thus, faults should not be injected there. However, a spurious injection was made there by the BM tool: the parameters of this function call were erroneously interpreted as distinct statements instead of an individual statement, due to the complexity of the expressions in input to *func_7()*. The BM tool handles the binary instructions for computing the function inputs (e.g., the calls to *func_5()* and *func_13()*, lines 1-16) *separately* from the instructions that invoke the function (i.e., the register *mov*es and the *branch* to *func_7()* in lines 17-19, in dark gray). A spurious fault is injected only in instructions 17-19, leaving the "useless" computation of function inputs in the loop, which is far from a realistic software fault that programmers would make.

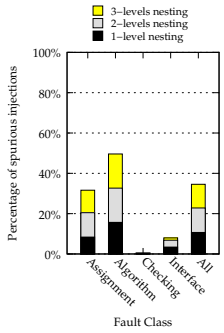
To avoid such spurious injections, the fault injector should take into account data dependencies between the instructions of the function call and the instructions that compute its inputs, and recognize all of



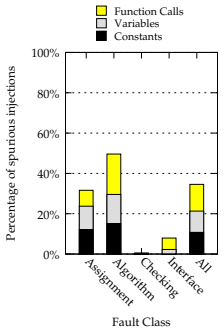
(a) Split by constraint.



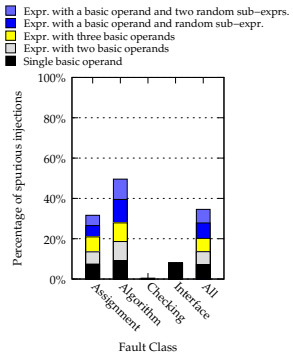
(b) Split by context.



(c) Split by nesting.



(d) Split by type of operand.



(e) Split by structure of expressions.

Fig. 6. Spurious injections.

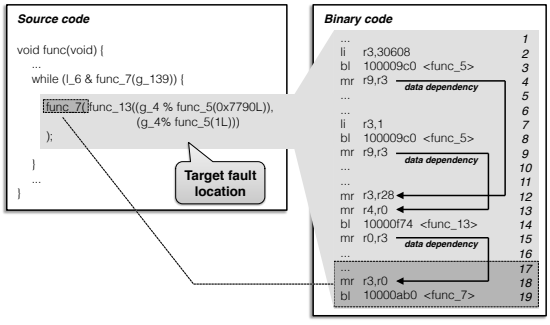


Fig. 7. Example of synthetic program causing a spurious injection (MFC fault type).

them as inter-dependent expressions. In the example of Fig. 7, the tool should avoid to inject a fault, since all the instructions in the *while* body block are all involved in the same function call, since the results of the expressions are all input parameters of the function call. It must be noted that, even if data-flow analysis is extensively used in compiler optimizations [43], to the best of our knowledge these techniques have still not been applied for accurate binary fault injection–we are currently investigating novel approaches based on the analysis of data dependencies [44]. We remark that having only a function call within a block is a corner-case issue, which can only be found with a systematic testing approach such as Faultprog. We found similar problems for the MIEB and MLPA fault type, since the tool does not correctly computes the number of statements within an *if* block, leading to spurious injections when an *if* block contains a loop or more than 5 statements.

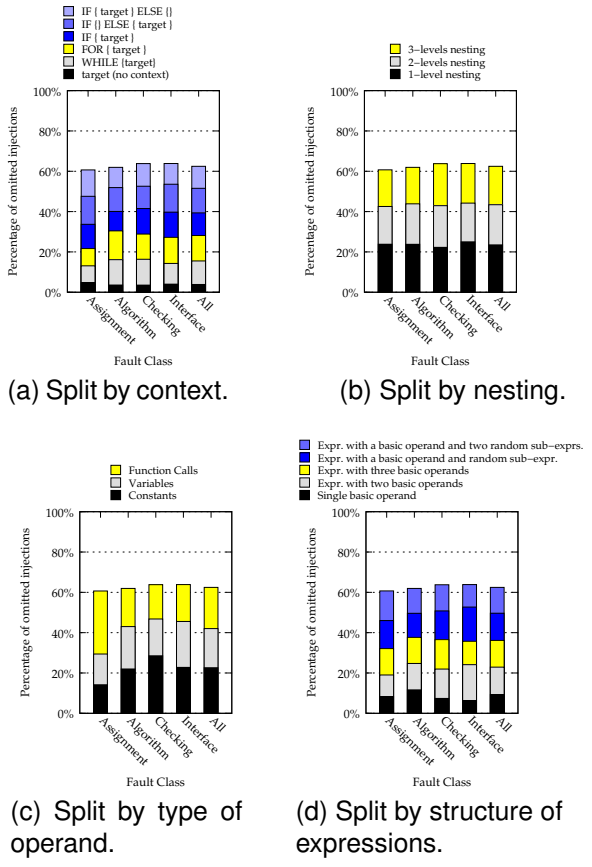
4.5.2 Omitted injections

Fig. 8 shows the distribution of omitted injections, again by splitting the bars with respect to Faultprog parameters (except for the lack of "constraint violations", that do not apply for "valid" test cases). The percentage of omitted injections is very high, for all ODC classes. However, these omitted injections cannot be attributed to a problem of a specific fault type or Faultprog parameter, since omitted injections occurred for every fault type, context, nesting, operand type, and expression structure (the bands in bars of Fig. 8a to 8d have about the same height in all cases).

From a closer analysis of omitted injections, we found that the root cause was the inaccurate identification of statement boundaries in the target fault location. As in the example of Fig. 7, the problem manifested for test cases with statements containing complex expressions, which were handled by csXception as a sequence of several small statements (instead of a unique, large statement). For this reason, csXception overestimated the number of statements in the blocks of the program, and (mistakenly) believed that constraints C02 or C09 were violated, leading to omitted injections. This problem also affected the Interface and Checking faults, even if these fault types do not require the C02 and C09 constraints: for these classes of faults, csXception did not recognize large statements that included several sub-expressions (e.g., for WPFV and WAEP, function calls with several sub-expressions in its parameters; for MLAC and MLOC, branch conditions with several boolean sub-expressions), since it did not parse these statements in their entirety, thus omitting to inject in them.

4.5.3 Incorrect injections

After identifying the test cases that did not cause omitted or spurious injections, we executed both their binary-mutated and source-mutated versions, and compared their execution. We then identified incorrect injections, i.e., test cases where a binary-mutant produced different results than the corresponding source-



(a) Split by context. (b) Split by nesting.

(c) Split by type of operand. (d) Split by structure of expressions.

Fig. 8. Omitted injections.

mutant. The percentage of incorrect injections was high across all fault classes, accounting for 60% of non-spurious, non-omitted injections. These incorrect injections were caused by another form of the statement boundary problem: the tool mutated only a *subset* of the binary instructions of the target statement.

For example, Fig. 9 shows an example for the MIA fault type. The BM tool was supposed to inject an MIA fault, by replacing the binary instructions of the *if* construct with *nops* (i.e., the instructions for evaluating the boolean expression in the *if* and the branch to the block after the *if*, highlighted in light gray in Fig. 9). To identify the target fault location, the BM tool first identifies the *branch* instruction related to the *if* (*beq-*, line 20 in the figure). Then, to identify the beginning of the *if* construct, it inspects backwards the binary instructions that precede the *branch*, by looking for either an assignment or a function call that denote the boundary with another statement. However, this technique did not work correctly for this program, as the binary instructions identified by the BM tool (dark gray in the figure) are only a subset of the target fault location. The BM tool stops the backward inspection in the middle of the *if* construct (line 10, at the end of the *func_12()* function call) rather than at the boundary of the statement (at line 4), thus not injecting the intended mutation in its entirety.

This incomplete mutation exhibits a different behavior than the corresponding source mutant: while the *branch* is removed from the program, the function call to *func_12()* is not removed, and affects its execution (e.g., through side effects). As discussed before, the BM tool should take into account the dependencies between binary instructions (in this case, the dependency is at the *mullw* instruction at line 18, which uses the results of the function calls) to accurately identify the boundaries of a fault location.

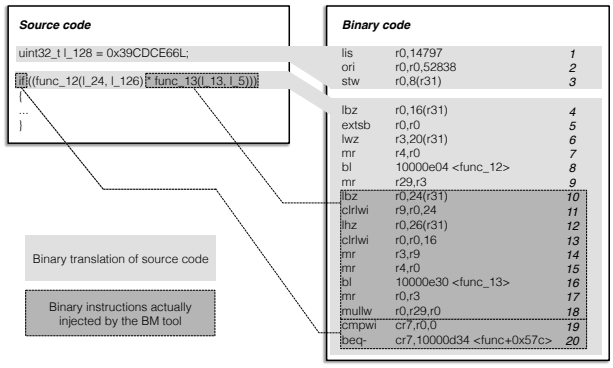


Fig. 9. Example of synthetic program causing an incorrect injection (MIA fault type).

4.5.4 *Impact of optimizations*

Finally, we consider the impact of compiler optimizations on the accuracy of Xception G-SWFIT. GCC provides three sets of optimizations ("levels"), which developers can selectively enable to reduce code size or to minimize compilation or execution time:

- **Basic optimizations**, which generate binary code with reduced size and improved performance, but without increasing the compilation time. This option is useful for many large server/database applications where memory paging due to larger code size is an issue.
- **Speed optimizations**, which include the optimizations of the previous level, and add global transformations that create faster, small code, but significantly increase the compilation time.
- **Speed-over-size optimizations**, which include the optimizations of the previous levels, and further improve performance but increase the size of binary code, such as loop unrolling, function inlining, and vectorization. They are used for applications with many floating point calculations or that process large data sets.

We apply again the Faultprog test-suite, by compiling test cases using these optimizations. However, we do not enable all optimizations at the same time (according to levels), since this approach would not easily allow to pinpoint the specific optimizations that interfere with injection accuracy. We hypothesize that most of the optimizations do not affect the code patterns involved in the fault model: therefore, we compile each test case several times and, at each

compilation, we enable only one optimization option, and keep disabled the remaining ones. This approach allows to easily identify specific optimization options that cause inaccuracies. In total, we consider 48 optimization options provided by the GCC compiler (documented in [45]), with the exception of loop unrolling and function inlining, which we already showed to be problematic in our previous work [15].

To analyze the impact of optimizations, we focus on test cases that were previously injected in a correct way. Test-cases are compiled with optimizations enabled, and are provided in input to the BM tool. The outcome is compared to the non-optimized test-case, to test whether the tool still injects correctly even when the binary code has been optimized. Compiler optimizations often merge the binary instructions of a group of source code statements: thus, debugging information now provides the binary-source mapping for the whole group of source code statements, instead of the mapping for individual statements. Therefore, we apply the BM tool on the whole group (both in the optimized and non-optimized versions), to check whether the number of injected faults is the same and, if so, whether all the execution results match.

Fig. 10 shows, for each optimization option, the percentage of test-cases that are still correctly-injected under that optimization option, where 100% means that the optimization had no impact on accuracy. We found that almost all optimizations have a negligible impact on accuracy, as the percentage of correct injections is about 97% for most of them (for readability, we do not show all of these optimizations with high accuracy in Fig. 10). However, there are three optimizations that decrease accuracy, namely *omit-frame-pointer*, *schedule-insns*, and *schedule-insns2*: they respectively change the convention for calling and returning from functions (by not using the *frame pointer* register), and reorder instructions (if they do not have dependencies) to avoid stalls that occur in pipelined CPUs. The former optimization hampers the identification of functions within the binary code (by changing the beginning and the ending of the binary translation of each function), thus it totally misleads the BM tool. In the other two optimizations, the compiler varies the order of binary instructions, thus leading to unexpected code patterns that were not considered during the development of the BM tool, and thus are not included in the implementation of the fault operators. Overall, the results of this impact analysis are encouraging for the adoption of binary mutation: (i) most of the optimizations do not interfere with the code patterns that are targeted by fault injection; and (ii) only few optimizations have to be dealt with, by extending the implementation of fault operators to cover their specific code patterns.

5 Related work

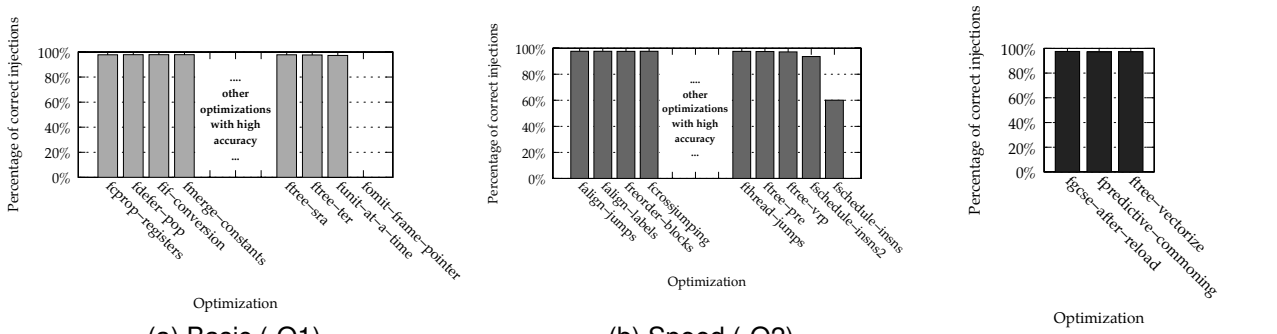
The injection of software faults has extensively been used for the evaluation of dependable systems. It worth to mention the use of fault injection to assess the likelihood of *high-severity* failures modes, such as: (i) data losses in a crash-tolerant file cache [8]; (ii) *fail-stop violations* in a transactional DBMS [7]; (iii) *error propagation* across components in a microkernel OS [9]. Another example is *dependability benchmarking* (e.g., OTS OSes, web servers, DBMSs) [10], which requires accurate injections to give a fair comparison.

Several binary-level fault injection techniques for OTS software have been proposed. Early SWIFI tools, such as FERRARI [46], NFTAPE [47], and Xception [39], adopted debug supports to introduce faults at run-time, by (i) inserting a *breakpoint* that triggers a handler when a target instruction is executed, and (ii) corrupting memory and CPU registers in the handler, by following a simple *bit-flip* fault model.

To better emulate software faults, researchers investigated more complex fault models. The FINE tool [48] proposed simple binary code mutations, by replacing machine instructions with *nops* and changing their destination operand. Subsequent studies, including Ng and Chen's [8], G-SWFIT [14], and LFI [49] refined the code patterns of binary mutations to reproduce more complex patterns, as discussed in § 2.

More recent studies leveraged virtualization and compiler technology to mutate binary code in a portable and easy-to-use way. XEMU [50] introduces binary mutations in the QEMU emulator, by extending its dynamic binary code translator. The program running in QEMU is first translated into an *intermediate representation* (IR), i.e., platform-independent instructions, and then translated again into native host instructions, using Just-In-Time compilation to keep low the translation overhead. In the middle of this process, XEMU injects mutations in selected IR code patterns (e.g., loads/stores, comparisons, assignments). In a similar way, ESI [51], LLFI [52], and EDFI [53] insert fault injection code at compile-time, by leveraging the LLVM compiler framework [54] to instrument the IR. If the source code of the target is available, these approaches can preserve the mapping between source code and IR code, and can thus achieve a high degree of accuracy. However, when only the binary code is available (as for OTS software), this solution cannot preserve the mapping with the source code, and the IR code has to be reversed from binary code [55], thus suffering from inaccuracies.

Several studies experimentally investigated the problem of the accuracy of binary-level fault injection, both for software faults [13], [14], [15] and hardware faults [16], [17]. Madeira et al. [13] analyzed how software faults translate into binary code, and how to emulate them through SWIFI with bit-flipping. They found that code patterns are in most cases too



(a) Basic (-O1). (b) Speed (-O2). (c) Speed-over-size (-O3).

Fig. 10. Impact of compiler optimizations.

complex for SWIFI techniques due to the semantic gap, and that inaccurate injections significantly impact on fault injection results. Other empirical studies analyzed the impact of the gap between the IR and machine code on SWIFI [17] and on fault activation [56]. However, these studies did not focus on the systematic *testing of BM tools* as in the present paper.

6 Conclusion

This paper presented the *Faultprog* approach for testing the accuracy of binary mutation tools, and a case study on a commercial tool. In this section, we discuss the limitations and the lessons learned of the study.

6.1 Limitations

To implement the Faultprog approach, we focused on the fault model of G-SWFIT, and made some design trade-offs for the Faultprog parameters (TABLE 3). The fault model and parameters restrict the synthetic programs that can be generated by our current implementation of Faultprog. We here discuss these restrictions, and suggest opportunities for extensions.

We focused the selection of basic operand types on local and global variables, constants, and function calls. In addition, basic operand types can be extended to include pointer variables, in order to test pointer-related fault types. In this paper, we did not consider pointer variables since this kind of operands are a known limitation of the BM tool that was identified in our previous study [15], and since G-SWFIT does not include specific fault types for pointers. Function calls could be extended to interact with the environment and have side effects that cannot be reverted, such as I/O interactions with the user. In this case, when comparing the executions of the binary-mutated and source-mutated programs, we should also compare the interactions with the environment (e.g., by tracing messages shown to the user) and use deterministic execution techniques [40] to make traces comparable.

We limited the "context" parameter by not mixing control-flow and loop constructs in the same test, and by only inserting the target statement in the body of control-flow constructs, but not in the controlling expression. These choices helped to avoid a significant increase of the number of test cases. Moreover, using the target statement as controlling expression would often lead to syntactically-invalid programs (e.g., an *if* cannot be within a controlling expression; injecting an MVIV could lead to an empty controlling condition). However, for specific fault types, Faultprog generates, through the "fault constraints" parameter, target statements within a controlling expression. For example, in order to test the fault constraint C06 for "missing assignment" faults, we negate the constraint and generate programs where the target statement (an assignment) is the controlling expression.

6.2 Lessons learned and generalization

The case study pointed out some challenging aspects for binary mutation. Besides the specific technical problems of a real BM tool, this experience points out the **importance of systematic testing of BM tools**. In fact, BM tools implement sophisticated fault models, such as G-SWFIT, and present an elaborated "input domain", i.e., binary programs generated by modern compilers. Thus, it is not sufficient anymore to test BM tools using few small programs [14], [57]. Instead, we need large test suites driven by coverage criteria and oracles, in order to identify specific areas of the input domain that need to be improved. In our case study, many inaccuracies were related to specific "fault constraints", that is, conditions that a program location has to satisfy to be eligible for fault injection. To test fault constraints, we generated synthetic programs that violated them, and uncovered problematic corner cases such as blocks with only one statement or with more statements than allowed.

Moreover, the most relevant issues are a consequence of the **more general problem of the "semantic gap"** between binary and source code, such as the heuristics for the identification of statement boundaries in binary code, which were based on simplified code patterns. The influence of the semantic gap depends on factors such as the source-level programming language, the reference hardware architecture,

and the fault model; however, similar issues will indeed affect other BM tools. This experience shows that the semantic gap is a challenge for injection accuracy and require robust techniques for binary analysis (e.g., using more complete code patterns to scan binary code, and looking at data dependencies).

Finally, we found that **many compiler optimizations have a negligible influence on BM accuracy**. For instance, changing the layout of data structures to increase the cache hit ratio will likely not affect the BM tool, since the code patterns that are typically targeted by fault models (such as assignments, control flow constructs, loops) will still hold even in the presence of optimizations. This result implies that BM is relatively insensitive to optimizations, and encourages its adoption in OTS software. However, our experiments also show that there are few optimizations that can impact on BM accuracy, such as those influencing function calls and instruction scheduling. Therefore, we suggest that BM tool developers should test each individual optimization, and that the BM tool should check the presence of problematic optimizations in a binary program (e.g., using reverse engineering techniques [58]) and to explicitly address these optimizations (e.g., by supporting alternative code patterns).

The case study aimed to show the feasibility and the usefulness of the Faultprog approach in a specific context, focusing on a specific architecture (PowerPC) and compiler toolchain (GCC). Thus, care must be taken to generalize the findings to other scenarios. However, the problems due to the semantic gap are not specific to this case study. Moreover, the case study targeted a widespread hardware architecture in embedded systems and representative of RISC architectures (such as ARM), and a popular, industrial-strength compiler toolchain, based on compiler techniques also adopted by other modern compilers [43]. Thus, similar issues and findings are likely to apply to other configurations. The possible differences with respect to different compiler versions or families are: (i) the use of different binary code patterns to translate programming constructs (such as loops, assignments, etc.) in the fault model, and (ii) a different set of optimizations. In these cases, developers should design the BM tool to address code patterns and optimizations of the target compiler that intersect with the fault model, and mutate them accordingly.

ACKNOWLEDGMENTS

This work has been supported by the CECRIS EU FP7 project (grant agreement no. 324334).

REFERENCES

[1] E. J. Weyuker, "Testing component-based software: A cautionary tale," *IEEE Software*, vol. 15, no. 5, 1998.
[2] J. Voas, "COTS software: the economical choice?" *IEEE Software*, vol. 15, no. 2, 1998.
[3] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Trans. on Soft. Eng.*, vol. 26, no. 9, 2000.
[4] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. on Soft. Eng.*, vol. 16, no. 2, 1990.
[5] J. Voas, "Certifying off-the-shelf software components," *IEEE Software*, vol. 31, no. 6, 1998.
[6] R. Natella, D. Cotroneo, and H. Madeira, "Assessing Dependability with Software Fault Injection: A Survey," *ACM Computing Surveys*, vol. 48, no. 3, 2016.
[7] S. Chandra and P. M. Chen, "How fail-stop are faulty programs?" in *Proc. FTCS*, 1998.
[8] W. T. Ng and P. M. Chen, "The Design and Verification of the Rio File Cache," *IEEE Trans. on Comp.*, vol. 50, no. 4, 2001.
[9] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," *IEEE Trans. on Comp.*, vol. 51, no. 2, 2002.
[10] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.
[11] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, 2005.
[12] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for Java," in *Proc. ESEC/FSE*, 2009.
[13] H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," in *Proc. DSN*, 2000.
[14] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Trans. on Soft. Eng.*, vol. 32, no. 11, 2006.
[15] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *Proc. EDCC*, 2012.
[16] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proc. DAC*, 2013.
[17] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Proc. DSN*, 2014.
[18] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *Proc. FTCS*, 1991.
[19] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *Proc. FTCS*, 1996.
[20] D. Cotroneo and R. Natella, "Fault Injection for Software Certification," *IEEE Security & Privacy*, vol. 11, no. 4, 2013.
[21] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification–A Concept for In-Process Measurements," *IEEE Trans. on Soft. Eng.*, vol. 18, no. 11, 1992.
[22] J. Durães and H. Madeira, "G-SWFIT fault emulation operators," http://wpage.unina.it/roberto.natella/misc/fault-emulation-annex-e0849s.pdf.
[23] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. on Soft. Eng.*, vol. 39, no. 1, 2013.
[24] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *Proc. QSIC*, 2003.
[25] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6, 2011.
[26] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *USENIX Security Symp.*, 2012.
[27] I. Hussain, C. Csallner, M. Grechanik, Q. Xie, S. Park, K. Taneja, and M. Hossain, "RUGRAT: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications," *Software: Pract. and Exp.*, 2014.
[28] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, 1998.
[29] D. Slutz, "Massive stochastic testing of SQL," in *Proc. VLDB*, vol. 98, 1998, pp. 618–622.

[30] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *ACM SIGPLAN Notices*, vol. 35, no. 1, 1999, pp. 1–13.
[31] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing system virtual machines," in *Proc. ISSTA*, 2010, pp. 171–182.
[32] Bryan Turner, "The Random C Program Generator," https://sites.google.com/site/brturn2/randomcprogramgenerator.
[33] GCC Online Documentation, "Options for Debugging Your Program or GCC," http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html.
[34] J. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. on Soft. Eng.*, vol. 18, no. 8, 1992.
[35] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 5, 2011.
[36] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. on Soft. Eng.*, vol. 40, no. 1, pp. 23–42, 2014.
[37] M. Papadakis, Y. Jia, M. Harman, and Y. LeTraon, "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique," in *Proc. ICSE*, 2015, pp. 936–946.
[38] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "How hard does mutation analysis have to be anyway?" in *Proc. ISSRE*, 2015.
[39] J. Carreira, H. Madeira, and J. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. on Soft. Eng.*, vol. 24, no. 2, 1998.
[40] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," in *Proc. ISSTA*, 2014.
[41] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia, "Assessing direct monitoring techniques to analyze failures of critical industrial systems," in *Proc. ISSRE*, 2014.
[42] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No PAIN, No Gain? The utility of PArallel fault INjections," in *Proc. ICSE*, 2015.
[43] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.
[44] A. K. Iannillo, D. Cotroneo, and R. Natella, "A Fault Injection Tool For Java Software Applications," Master's thesis, Federico II University of Naples, Italy, 2013.
[45] GCC Online Documentation, "Options That Control Optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.
[46] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. on Comp.*, vol. 44, no. 2, 1995.
[47] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer, "A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," in *Proc. ICPDS*, 2000.
[48] W.-I. Kao, R. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Trans. on Soft. Eng.*, vol. 19, no. 11, 1993.
[49] P. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM Trans. on Comp. Sys.*, vol. 29, no. 4, 2011.
[50] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "XEMU: an efficient QEMU based binary mutation testing framework for embedded software," in *Proc. EMSOFT*, 2012.
[51] U. Schiffel, A. Schmitt, M. Susskraut, and C. Fetzer, "Slice your bug: Debugging error detection mechanisms using error injection slicing," in *Proc. EDCC*, 2010.
[52] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," *Proc. SELSE*, 2013.
[53] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments," in *Proc. PRDC*, 2013.
[54] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, 2004.
[55] C. Lattner, "Intro to the LLVM MC Project," http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html.
[56] E. van der Kouwe, C. Giuffrida, and A. S. Tanenbaum, "Evaluating Distortion in Fault Injection Experiments," in *Proc. HASE*, 2014.
[57] A. Jin and J. Jiang, "Fault Injection Scheme for Embedded Systems at Machine Code Level and Verification," in *Proc. PRDC*, 2009.
[58] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," in *Proc. NDSS*, 2011.

Domenico Cotroneo (Ph.D.) is associate professor at the Federico II University of Naples. His main interests include software fault injection, dependability assessment, and field-based measurements techniques. He has been member of the steering committee and general chair of the IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), PC co-chair of the 46th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN), and PC member for several other scientific conferences on dependable computing including SRDS, EDCC, PRDC, LADC, and SafeComp.

Anna Lanzaro (Ph.D.) is a postdoctoral researcher at CINI and at the Federico II University of Naples, Italy. Her research interests are on the dependability assessment of safety-critical systems, on hardware and software fault injection, and on the dependability of multicore and virtualization technologies. She was recipient of the Best Presentation award at the 9th European Dependable Computing Conference (EDCC 2012) for a previous version of this paper.

Roberto Natella (Ph.D.) is a postdoctoral researcher at the Federico II University of Naples, Italy, and co-founder of the Critiware s.r.l. spin-off company. His research interests include dependability benchmarking, software fault injection, and software aging and rejuvenation, and their application in operating systems and virtualization technologies. He has been involved in projects with Finmeccanica, CRITICAL Software, and Huawei Technologies. He contributed, as author and reviewer, to several journals and conferences on dependable computing and software engineering, and he has been in the steering committee of the workshop on software certification (*WoSoCer*) held with recent editions of the ISSRE conference.