Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 17-008

Toward Rigorous Object-Code Coverage Criteria

Taejoon Byun, Vaibhav Sharma, Sanjai Rayadurgam, Stephen McCamant, Mats Heimdahl

June 13, 2017

Toward Rigorous Object-Code Coverage Criteria

Taejoon Byun, Vaibhav Sharma, Sanjai Rayadurgam, Stephen McCamant, Mats Heimdahl
Department of Computer Science & Engineering, University of Minnesota, Minneapolis, MN, USA
Email: [taejoon, vaibhav]@umn.edu, [rsanjai, mccamant, heimdahl]@cs.umn.edu

***Abstract*—Object-branch coverage (OBC) is often used as a measure of the thoroughness of tests suites, augmenting or substituting source-code based structural criteria such as branch coverage and modified condition/decision coverage (MC/DC). In addition, with the increasing use of third-party components for which source-code access may be unavailable, robust object-code coverage criteria are essential to assess how well the components are exercised during testing. While OBC has the advantage of being programming language independent and is amenable to non-intrusive coverage measurement techniques, variations in compilers, and the optimizations they perform can substantially change what is seen as an object branch, which itself appears to be an informally understood concept. To address the need for a robust object coverage criterion, this paper proposes a rigorous definition of OBC such that it captures well the semantic of source code branches for a given instruction set architecture. We report an empirical assessment of these criteria for the Intel x86 instruction set on several examples from embedded control systems software. Preliminary results indicate that object-code coverage can be made robust to compilation variations and is comparable in its bug-finding efficacy to source level MC/DC.**

I. Introduction

Object-Branch Coverage (OBC) is a structural coverage criterion at the object-code level which requires that a test suite executes both branches of conditional jumps [1]. It is frequently used as a measure of the thoroughness of tests suites in safety critical domains [2], [3], augmenting or substituting source-code based structural criteria such as (source code) branch coverage and modified condition/decision coverage (MC/DC). OBC may also be used in aeronautics, as DO-178C [**?**] standard mandates the test coverage measurement to be performed on the binary level when traceability to the source code cannot be established. In addition, with the increasing use of third-party components for which source-code access may be unavailable [4], [5], robust object-code coverage criteria are essential to assess how well the components are exercised during testing.

While OBC has the advantage of being programming language independent and is amenable to non-intrusive coverage measurement techniques, variations in compilers and the optimizations they perform can substantially change the structure of the object code, the number of branches, as well as how branching is implemented; optimizations – depending on the compiler configuration – can significantly affect the fault finding capability of test-suites providing OBC coverage.

We have in our work empirically assessed the effect of different compilers and compiler optimization settings on the fault-finding effectiveness of OBC. We observed a drop in mutation score by up to 52% depending on what compiler optimizations were used; we measured a mutation score of 53% when the code was compiled with no optimization to a mere 28% with the most aggressive optimization. This sensitivity to the structure of the generated code has been recognized in the literature [6]–[8] and had an effect that complicates the certification process [7] and may discourage testers from relying on object-level coverage criteria [8]. These limitations of OBC leaves the community without an effective means of measuring test adequacy over object code.

To address the need for a robust object coverage criterion, this paper proposes a rigorous definition of OBC (called Flag-Use Object Branch Coverage); robust in this context implies a criterion that is effective in fault finding as well as insensitive to the structure of the object-code code over which it is measured. Modifications to the existing definitions of OBC are necessary since compilers perform optimizations that might, for example, replace a jump instruction with a conditional move instruction; thus, in effect, eliminating a branch point. With our definition of Flag-Use Object Branch Coverage, we aim for a criterion that captures the intuition behind source code coverage criteria such as branch coverage, decision coverage, and modified condition/decision coverage, but is measured over the generated code in the instruction set for a given architecture (in this paper we work over the x86 instructions set). Flag-Use Object Branch Coverage extends OBC to include not only jump instructions, but many other instructions involved in conditional – thus branching – behavior.

To provide an initial assessment of our proposed Flag-Use Object Branch Coverage criterion, we report on an empirical assessment of OBC and Flag-Use Object Branch Coverage for the Intel x86 instruction set on a set of five case examples from the embedded software domain. The preliminary results indicate that object-code coverage can be made more robust to compilation variations through a broader definition of what constitutes a branch at the object-level – Flag-Use Object Branch Coverage. A comparison with conventional OBC and MC/DC showed that Flag-Use Object Branch Coverage is comparable in its fault-finding efficacy to source level MC/DC. There are, however, additional issues to address, for instance, Flag-Use Object Branch Coverage is sensitive to the structire of the object-code – albeit far less so than OBC – and the fault-finding ability of test suites satisfying Flag-Use Object Branch Coverage (as well as MC/DC for that matter) is not as strong as one would like.

II. Background

A. Object Code vs. Source Code

Source-level coverage criteria is defined over the structure of the source code. Branch coverage, for instance, requires a test suite to exercise each side of the control structures in a source code, and is usually defined as all-edge coverage of the control flow graph (CFG) [9]. When a control-flow graph is constructed in the source-level, each conditional statement in the source code constitutes a conditional node while each non-conditional statement constitutes an internal node. Figure 1a shows such CFG, constructed from the C code in Listing 1.

Listing 1: Illustrative example, source code in C

```
1     static char msg[2][5] = {"pass", "fail"};
2     int pass;
3     if (grade=='d'||grade=='f') {
4         pass = 0;
5     } else if (grade=='a' || grade=='b' ||
          grade=='c') {
6         pass = 1;
7     } else { pass = -1; }
8     return pass ? msg[pass] : msg[0];
9   }
```

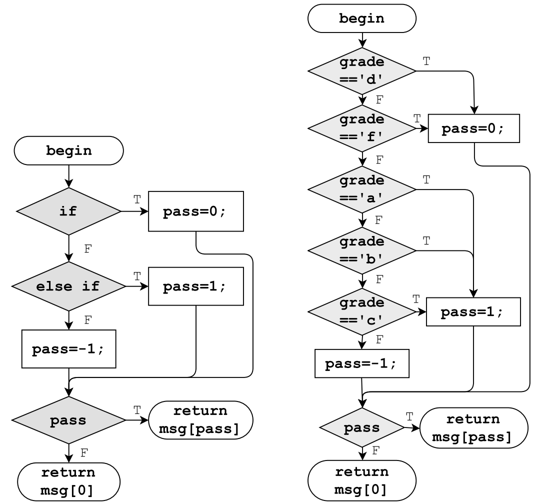
The CFG constructed from the source code, however, is coarse-grained and does not fully represent the details of an actual control flow in the machine-code level. This is because of the short-circuit operations supported by C (Section 6.5.13 of the latest C standard [10]) that evaluates second condition only when the decision outcome cannot be determined by evaluating the first condition. In effect, it allows a faster computation for complex boolean decision because the whole expression need not be evaluated when the outcome is obvious by evaluating a part of it. On CFG, it has an effect of splitting a conditional node of a complex boolean expressions into multiple conditional nodes. CFG edge coverage criterion applied on object-code CFG is thus stronger than the one applied on source-code CFG, because the test suite required by the former subsumes the latter.

If we take short-circuit evaluation into account, the actual CFG in the machine-code level looks like Figure 1b. The `if` node in Figure 1a is broken down into two conditional nodes in Figure 1b such that the second condition is not evaluated when the first condition evaluates to true. This effect can also be seen in the corresponding object code in Listing 2 where each conditional jump instruction (highlighted with shaded marker) corresponds to the conditional node in Figure 1b.

The logic of short-circuit evaluation, however does not apply when a compiler optimizes, as will be described in the following subsection.

B. An Illustrative Example

Listing 1 shows a simple function written in C that returns a string literal based on the given character argument. The function is designed to return `"pass"` when the grade is `a`, `b`, or `c`, and return `"fail"` when it is `d` or `f`. This code, however, is faulty for the cases when an invalid input is given. When `grade` is `e`, for example, the local variable `pass` is assigned



(a) CFG of the source code (b) CFG of the object code

Fig. 1: Control flow graphs of example code in Listing 1

to `-1` in line 8, which lets the ternary operator to take the true side, and thus the address of `msg[-1]` will be returned in line 9. This bug is trivial in structural-testing perspective; satisfying even the simplest form of source-level coverage criterion can reveal this fault. Statement coverage will enforce a test suite to cover every statement and thus will require at least one test case that exercises `pass=-1`. Branch coverage will insist on taking every side of if-else statement and thus shall require a test case that visits the else part.

Listing 2: *x86* code compiled with `-O0`

```
1   8048439: cmpb   $0x64,-0x14(%ebp)
2   804843d: je     8048445          // grade=='d'
3   804843f: cmpb   $0x66,-0x14(%ebp)
4   8048443: jne    804844e          // grade!='f'
5   8048445: movl   $0x0,-0x4(%ebp)  // pass:=0
6   804844c: jmp    8048470 // jump to return
7   /* else if (grade=='a'||...||grade=='c') */
8   804844e: cmpb   $0x61,-0x14(%ebp)
9   8048452: je     8048460 // grade=='a'
10  8048454: cmpb   $0x62,-0x14(%ebp)
11  8048458: je     8048460 // grade=='b'
12  804845a: cmpb   $0x63,-0x14(%ebp)
13  804845e: jne    8048469 // grade!='c'->else
14  8048460: movl   $0x1,-0x4(%ebp)  // pass:=1
15  8048467: jmp    8048470
16  /* else         (pass:=-1) */
17  8048469: movl   $0xffffffff,-0x4(%ebp)
18  ...
```

When it comes to OBC, however, this simple bug may or may not be caught depending on the compiler optimization applied when generating the object code. Listing 2 and 3 show two different versions of the object-code disassembly compiled with GCC 5.4 on Ubuntu running on Intel x86-64 machine, where no optimization (`-O0`) is applied on the former and the size optimization (`-Os`) is applied on the latter. The first seven-digit number is the instruction pointer address, the following string highlighted is the instruction mnemonics, and the rest are the operands.

TABLE I: Test cases required for each criterion

Test case	grade	pass	Branch	MC/DC	OBC -O0	OBC -Os
t_1	a	1	✓	✓	✓	✓
t_2	b	1		✓	✓	
t_3	c	1		✓	✓	
t_4	d	0	✓	✓	✓	✓
t_5	**e**	**-1**	✓	✓	✓	
t_6	f	0		✓	✓	

Listing 3: *x86* code compiled with -Os

```
1  8048456: mov  $0x804a01c,%eax// %eax:=msg[0]
2  804845b: mov  %esp,%ebp
3  804845d: mov  0x8(%ebp),%edx // %edx:=grade
4  /* if (grade == 'd' || grade == 'f') */
5  8048460: mov  %dl,%cl           // %cl:=grade
6  // ASCII('d')=0x64, ASCII('f')=0x66,
7  // 'f'^0xfffd='d', 'd'^0xfffd='d'
8  8048462: and  $0xfffffffd,%ecx
9  8048465: cmp  $0x64,%cl  // 'd', grade
10 8048468: je   804847e    // %cl=='d'->return
11 /* else if (grade=='a'||...||grade=='c') */
12 /* else */
13 804846a: sub  $0x61,%edx  // %edx=grade-'a'
14 804846d: cmp  $0x3,%dl    // CF=%edx<3?1:0
15 8048470: sbb  %eax,%eax   // %eax:=CF?-1:0
16 8048472: and  $0x2,%eax   // %eax:=CF?2:0
17 8048475: dec  %eax        // %eax:=CF?1:-1
18 /* return pass ? msg[pass] : msg[0]; */
19 // %eax:=5*%eax
20 8048476: imul $0x5,%eax,%eax
21 // %eax:=msg+%eax
22 8048479: add  $0x804a01c,%eax
23 804847e: ...
```

It is notable at a first glance that the number of object-code branches differ dramatically between Listing 2 and 3 – the former has five branches (je and jne) while the latter only has one. The first branch in line 3 of Listing 2 is abstracted using and instruction in line 7 of Listing 3; as a consequence of the instruction, f becomes d since 0xfffffffd $\wedge$ 0x66 $=$ 0x64, while d remains the same as 0xfffffffd $\wedge$ 0x64 $=$ 0x64. The three jump instructions in line 10, 12, and 14 of Listing 2 are translated without using any conditional jumps, which we will not explain for a lack of space.

From the fact that there are significantly less branches in the optimized code, one can guess that the efficacy of OBC might not be as good with optimized code compared with unoptimized code. To see the effect of the optimization on the fault-finding efficacy of OBC, we illustrated in Table I the test cases that are necessary to satisfy branch coverage, MC/DC, OBC on -O0 code and OBC on -Os code, respectively.

Table I enumerates six test cases for each representative input value. The third column shows the value of the local variable pass after an execution of each test case. The following four columns mark on the minimal set of test cases that is necessary to satisfy the corresponding coverage criterion. Our measure of the efficacy of a coverage criterion for this particular example is whether a criterion selects the test case that reveals the fault, which is t_5.

Branch coverage requires either one of $\{t_4, t_6\}$ to cover if side, one of $\{t_1, t_2, t_3\}$ to cover the else if side, and t_5 to cover else side. It thus is capable of revealing the bug in line 8 of Listing 1. MC/DC is even stronger than branch coverage since it requires each condition in decision to exercise both true and false, and also to show that it independently affects the decision outcome. As a result, all the six test cases has to be selected in order to satisfy MC/DC, also revealing the fault as a result. OBC on -O0 code is similar to MC/DC in this case; because of the conditional jumps that asserts each condition to take both the outcomes, all six test cases are included.

When OBC is applied on the -Os code, on the other hand, it only requires two test cases because they suffice to cover the only conditional jump in line 9 of Listing 3. OBC will require either one of $\{t_4, t_6\}$ to cover the true side of the jump, and one of $\{t_1, t_2, t_3, t_5\}$ to cover the false side of the jump. As a result, one can go away without including t_5 and still satisfy OBC on this executable. We believe that this susceptibility is not desirable as a coverage criterion.

III. Flag-Use Object Branch Coverage

As demonstrated in Section II-B, OBC based on conditional jumps alone is susceptible to compiler optimization as the branches in the source code may be translated to other conditional instructions. For instance, sbb (subtract with borrow) instruction in line 14 of Listing 3 is a conditional instruction that depends on the value of carry flag (CF); by default, it subtracts the second operand from the first operand, but when the carry flag is set, it subtracts one more. If both these possibilities must indeed be exercised for meeting a coverage criterion, then a test case such as t_5 would be included in every test suite satisfying that criterion, thereby triggering the fault. We capture this intuition in the form of a stronger coverage criterion that we call **Flag-use OBC**. Informally, flag-use OBC, requires a test suite to exercise all disntict behaviors of each conditional instruction that depends on flag usage.

A. Flag-use Object Branch Coverage

Definition 1: Conditional Instruction: Any instruction whose behavior changes based on its execution context, not including its direct inputs, is said to be a conditional instruction.

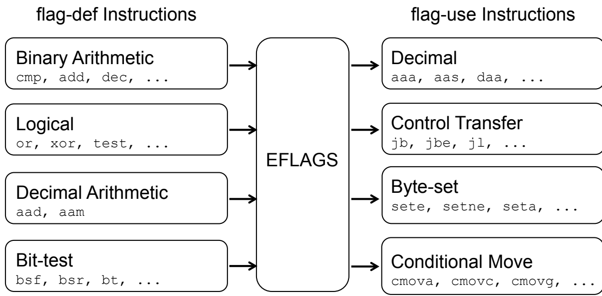
Definition 2: Object Branch Coverage: Object branch Coverage is defined as coverage of conditional behaviors of instructions at the object-level.

While Definition 1 includes instructions such as conditional branches, it also includes instructions such as the IN instruction, which is used for reading data from a I/O port. We use a definition of OBC that is closer to source-like coverage, and discards binary-level checks that were not added by the programmer. Hence, we introduce Flag-Use coverage.

Definition 3: Flag-Use Coverage: Flag-Use Coverage is defined as coverage of conditional behaviors of instructions that either (1) read non-system flags from the EFLAGS register, or (2) are conditional branch instructions.

Predicated instructions such as cmovne, and setne are just as crucial to establishing coverage as conditional branch instructions like jne. Definition 3 is a usable definition of OBC, since it removes dependence of OBC on system state, and allows

Fig. 2: Instructions that define and use conditions



coverage obligations to reflect source-like obligations at the object level. We describe our identification of flag-use X86 instructions in the following subsection.

B. Identifying X86 conditional instructions

As shown in Figure 2, across different X86 instruction categories, some X86 instructions write to the `EFLAGS` register, and other instructions read from it. We identified flag-use X86 instructions in three steps. First, we obtained a list of instruction byte sequences that captured the behavior of every X86 32-bit instruction. Next, we performed automated classification of the behavior of every instruction byte sequence as exhibiting conditional behavior or not, and recorded the source of conditional behavior. Finally, we manually combined the classification output with the Intel IA-32 Architecture manual [11], and obtained a list of instructions along with the source of each instruction's conditional behavior. We describe these three steps as follows:

1) Instruction Set Exploration: We obtained instruction byte sequences by exploring the X86 instruction set, as performed by the PokeEMU tool described by Martignoni et al. [12]. The PokeEMU framework generates high-coverage test cases for an emulator and allows those tests to be run on a different emulator or a real machine for comparison. Martignoni et al. used the PokeEMU framework to compare a low fidely emulator (QEMU [13]) with a high fidelity emulator (Bochs [14]). Similar to PokeEMU, we performed an exploration of the X86 instruction set by symbolically executing the instruction decoder of Bochs with the first three bytes of the instruction byte sequence set to be symbolic. The symbolic execution was performed using FuzzBALL [15], a binary symbolic execution tool for machine code. This gave us a list of 76510 candidate byte sequences which are valid instructions as per the Bochs instruction decoder. While some instruction prefixes (such as `rep`) allow further exploration of conditional behavior in instructions, other prefixes (such as `lock`, `gs`) do not. Using the `lock` prefix does not cause any change in instruction behavior in a single-threaded context. Using segment override prefixes requires segment registers to be setup correctly before execution of the instruction without giving the instruction any additional conditional behavior. we chose to ignore a total of seven instruction prefixes (`lock`, `cs`, `ss`, `ds`, `es`, `fs`, `gs`). For every byte sequence, we first checked if its disassembled string representation contained any of these seven prefixes as a substring, and removed corresponding prefix bytes from the instruction byte sequence, if it did. Removal of the prefix byte(s) could cause a byte sequence to become equal to a previously decoded byte sequence. We saved byte sequences into a hashtable, and discarded a byte sequence if it was decoded previously. This reduced our 76510 byte sequences to 45311 unique byte sequences.

IV. Experiment

In the experiment, we first present a study on the sensitivity of object-level coverage criteria to the compiler configuration. We also compare the effectiveness of Flag-Use OBC to the conventional OBC under the same configuration to show the gain in fault-finding effectiveness and robustness to compiler configuration. In summary, we aim to answer the following research questions:

- **RQ1**. Does the compiler configuration affect the fault-finding efficacy of a test suite that satisfies object-level coverage criteria?
- **RQ2**. Is a test suite satisfying Flag-Use OBC more effective and robust to program restructuring than a suite that satisfies conditional OBC?
- **RQ3**. Is OBC or Flag-Use OBC more effective than MC/DC in terms of fault-finding effectiveness?

To answer these research questions, we generate test suites for each configuration – 1) a choice of compiler, 2) an optimization option, and 3) a coverage criterion – and compare the mutation scores among them. Since object-level coverage criterion base its measure on the object code, the same program compiled with different compiler or compiler option requires different test suites to fulfill its coverage obligations.

A. Experiment Setup Overview

We have performed the experiments on five industrial systems developed by Rockewell Collins (*Cruise Controller*, *Microwave*, and *Microwave (C)*), University of Minnesota (*Infusion Pump*), and NASA (*Docking Approach*). All of the case examples were modeled using Stateflow [16], were translated to Lustre synchronous programming language [17] to utilize our existing automation, and finally translated to C using Verimag Lustre V6 Tool Chain [18]. *Microwave*, however, was also written in C from the same specification with which the Stateflow version was implemented. We prepared two different versions of *Microwave* example to study the effect of structural difference in the object code introduced by the structural difference in the source level.

For each system under test, we performed a sequence of steps illustrated in Figure 3 to compare the fault-finding effectiveness of each test suite. A brief overview of each step is as follows:

1) We apply mutation into the source program and generate 250 mutants. (Section IV-B).
2) We construct a large pool of test cases called *master suite*, using various test generation methods (Section IV-C).

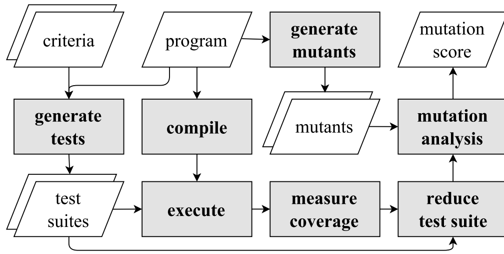


Fig. 3: Experiment Process

3) We compile the program for each choice of compiler configuration (Section IV-D).
4) We execute each test case on the compiled program, measure OBC, and save the execution trace for a later mutation analysis (Section IV-E).
5) The reduction step selects minimal test suites while preserving the highest achievable coverage of a given criterion. (Section IV-C).
6) We measure the mutation score by comparing the execution trace using output-only oracle.

B. Mutant Generation

For each case example, we created 250 mutants by introducing a single fault in the correct Lustre implementation. For *Microwave (C)* example where the source code is written in C, we used the same set of mutants generated from Each fault was seeded by either replacing an operator or variable in the program, or inserting a new operator. The mutation operators used this study is typical, and discussed in length by Rajan *et al*. [19]. We did not bother to remove the equivalent mutants in this study, since the definitive score does not purpose of the study is to compare the relative efficacy.

C. Test Generation and Reduction

When comparing the effectiveness of coverage criteria empirically, one should be cautious on controlling variables other than coverage criterion itself, including test generation method. If different test generation methods are used while comparing coverage criteria, the generation method itself can also affect the efficacy of a test suite, and thus may lead to an invalid comparison [20]. To avoid this issue, we derive the test suites for each criterion from the same *master suite*. We assume the master suite to be a huge pool of test cases of varying quality, that master suite alone can achieve a good fault-finding efficacy and a high coverage for any given coverage criterion.

In practice, we construct the master suite by combining the test cases generated by two available test generation methods – random generation and coverage-guided test generation for MC/DC and source-level branch coverage. For coverage-guided test generation, we utilized counterexample-based test generation [21] using model checker, which guarantees that the test suite achieves the highest achievable branch coverage and MC/DC coverage in the source level.

To enable comparison among coverage criteria, we *reduce* the master suite with respect to each coverage criterion while maintaining the coverage achieved. We randomly pick a test case from the master suite, measure the coverage, and add the picked one only if it improves the coverage. We repeat this process until the maximum coverage is reached, and the resultant test suite is thus minimal in size while preserving the coverage of the master suite. To prevent us from selecting test cases that are exceptionally good or bad, we construct 40 different test suites per each configuration.

D. Compilation

Several works have shown for MC/DC coverage criterion that the structure of program can affect the effectiveness of MC/DC suite dramatically [22], [23]. The issue remains the same in the object code level, except that there is another step that affects the program structure – comiplation. On top of the structural difference introduced by developers in the source level, compiler introduces another layer of complexity through its translation algorithm and transformation mechanisms, or compiler optimizations.

To see the effect of compiler on the effectiveness, we used three different optimizing compilers as follows:

- GNU Compiler Collection (GCC) [24] is the standard compiler adopted by many Unix-like operating systems.
- Clang [25] is a compiler front that uses LLVM as its back end. It is a more recent compiler than GCC and it claims to have a better optimizer.
- CompCert [26] is a formally verified optimizing compiler targeted especially for critical embedded software.

To study the effect of compiler optimization on the effectiveness, we compare among five of the most common optimization options: `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. The convention is that `-O0` does not apply any optimization while `-O1` applies all easy-to-apply optimizations, followed by `-O2` and `-O3` that optimizes even further, and `-Os` that optimizes for a minimal code size [27]. Although all the compilers we compared provides the five optimization levels as an interface for controlling the optimization levels, the specific optimizations that each compiler implements are dependent on compiler. CompCert, for instance, does not apply more optimizations when `-O2` or `-O3` is given.

E. Measuring OBC

We implemented a tool chain using Pin [28], which is a dynamic instrumentation system that allows users to write their own tool (called *Pintool*) to perform a binary analysis task. Our *Pintool* instruments tracing code in the instruction-level granularity for each conditional instruction we identified in Section III-B. When executed with *Pin*, the *Pintool* instruments code before or after the execution of each conditional instruction. This code then prints a trace of branch encounters where each entry includes instruction pointer addresses, instruction mnemonic, and the side of the branch taken. After the coverage trace is obtained, we analyze it against a disassembly of the object code using a separate tool and obtain a quantitative measure.

V. RESULT

For each case example described in Section IV-A, we measured the mutation score while varying three parameters of our interest – 1) compiler, 2) optimization level, and 3) the coverage criterion to guide the test suite construction.

The result is presented in Table II as box-and-whisker plots. Each row of seven plots corresponds to the same case example, and each column stands for the coverage criterion applied to construct the test suites, along with the choice of compiler used to generate the object code for the object-level coverage criteria. For each case example and a choice of compiler, we placed the result of conventional OBC and Flag-Use OBC side by side for an easier visual comparison between the two.

In each plot, x-axis corresponds to the level of compiler optimization applied while generating object code, which is labeled as `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, respectively. Y-axis shows the percentage of mutants killed using the output-only oracle, comparing only the output variables between the original and the mutant. The median of the data set is depicted as a line with dots around, and the mean value is represented as dashed lines. The boxes represent the second and the third interquartile range (IQR), whiskers show the range of data within $1.5 \times IQR$ from the upper and lower quartile, and small circles mark the outliers that lie beyond $1.5 \times IQR$ from the second and the third quartiles.

As illustrated in Table II, the mutation scores vary from 1% to 72% depending on the case example. Note that the score itself is not a definitive measure of the efficacy of OBC, and thus shall not be interpreted as its absolute value for the following reasons: 1) we did not eliminate the equivalent mutants, and 2) our use of output-only oracle can diminish the fault-finding efficacy when a fault inside a program cannot be observed through output variables. Among the two factors, the use of output-only oracle resulted in significantly low mutation scores for some case examples such as *Cruise Controller*. In this particular case example, there is only one output variable of boolean type, that a fault induced by a mutation is extremely difficult to be detected. Despite this effect, we believe that the difference induced by compiler configuration is still well manifested to enable a relative comparison among the groups.

A. RQ1: Sensitivity of Object-Level Criteria to Compilation

We hypothesize that a higher compiler optimization will decrease the fault-finding effectiveness of an OBC suite because a higher optimization may not preserve all the conditional jumps for the sake of saving computational cost. In other words, we hypothesize that:

$\mathbf{H_1}$: The compiler configuration affects the fault-finding effectiveness of a test suite that satisfies OBC.

A corresponding null hypothesis of H_1 is as follows:

$\mathbf{H_0}$: The compiler configuration has no effect on the fault-finding effectiveness of a test suite that satisfies OBC.

To test this hypothesis, we performed a statistical test using Welch's Analysis of Variance (ANOVA) – a statistical hypothesis test to determine if the differences among group means are significant while the variance among groups are heterogeneous. For each combination of case example and the choice of compiler, we treated the optimization level as an independent categorical variable and the mutation score as a dependent variable.

Table III shows the result of ANOVA F-test from the data illustrated in Table II, where the data set is mutation scores across different compiler optimizations. We report the F-value along with the p-value, where F-value is the ratio of between-group variability to the within-group variability. A corresponding p-value shows the likelihood of such variance being manifested under the null hypothesis. We reject H_0 – that compiler configuration has no affect on mutation score – when p-value is less than 0.01 threshold.

As we can observe from the figures in Table II, the statistical analysis also suggests that H_0 does not hold in most cases. One of the case that most dramatically illustrates the effect of optimization is *Microwave* compiled with Clang (Figure (2, 4) in Table II). The median mutation score dropped from 53% with `-O0` to a mere 28% with `-O3`, illustrating the extent the optimization can affect the efficacy of OBC. This susceptibility is also captured in the F-test where the F-value is 198.25 with a corresponding p-value of < 0.01. Flag-Use OBC, on the other hand, showed a significant improvement both in terms of the mutation score and the between-group variability, as can be seen in Figure (2, 5) in Table II. According to the F-test, however, the improvement was not significant enough to conclude that compiler optimization does not affect the mutation score; the F-value of Flag-Use OBC with the same case example decreased to 24.03, but the p-value still is lower than 0.01, which lead us to reject H_0.

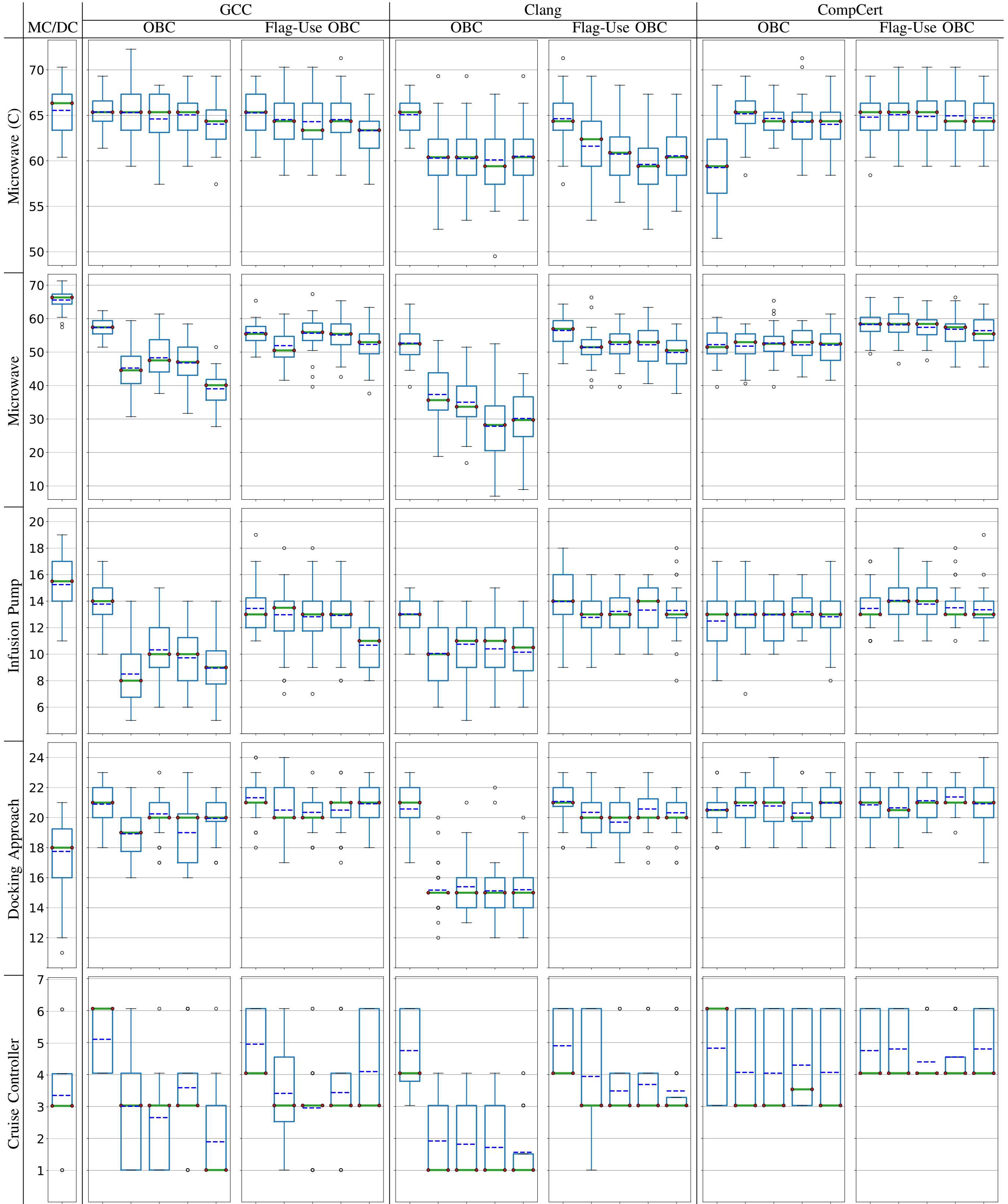
Based on the 0.01 threshold of p-value, we reject the null hypothesis in 11 out of 15 cases for OBC, concluding that the efficacy of OBC is highly sensitive to compiler configuration. This sensitivity decreased in most cases with Flag-Use OBC; the F-values – which represent the ratio of between-group variability to the within-group variability – decreased significantly in 12 out of 15 cases (highlighted with bold-faced font). Despite this improvement, however, the variability was still significant enough to reject H_0 in 10 out of 15 cases. In conclusion, Flag-Use OBC is also sensitive to compiler configuration, albeit far less so than conventional OBC.

B. RQ2: OBC vs. Flag-Use OBC

Figures in Table II visually compare the trend in mutation scores between OBC and Flag-Use OBC. It can be seen throughout most of the cases that Flag-Use OBC outperforms OBC with a higher average mutation score and a smaller variance across optimization levels.We further compared the populations of mutation scores between the two, specifically to compare the average mutation score – by comparing the mean – and the variance – by comparing the standard deviation within group.

Table IV shows the difference in average mutation scores and the difference in standard deviation between OBC and Flag-Use OBC for each configuration of case example, compiler, and optimization level. A positive difference in means

TABLE II: Comparisons of fault-finding effectiveness



Each row is for one case example as labeled. In each graph, x-axis is labeled as `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. Y-axis is the mutation score (%).

TABLE III: Analysis of variance across optimization levels

Case Example	Compiler	OBC			Flag-Use OBC		
		F-value	p-value	H_0?	F-value	p-value	H_0?
Microwave (C)	GCC	4.94	< 0.01	reject	5.87	< 0.01	reject
	Clang	**37.48**	< 0.01	reject	**25.99**	< 0.01	reject
	CompCert	**42.17**	< 0.01	reject	**1.33**	0.27	accept
Microwave	GCC	**101.83**	< 0.01	reject	**12.65**	< 0.01	reject
	Clang	**198.25**	< 0.01	reject	**24.03**	< 0.01	reject
	CompCert	**1.39**	0.24	accept	**0.37**	0.83	accept
Infusion Pump	GCC	**35.84**	< 0.01	reject	**14.40**	< 0.01	reject
	Clang	**31.45**	< 0.01	reject	**0.98**	0.42	accept
	CompCert	0.17	0.95	accept	0.23	0.92	accept
Docking Approach	GCC	**28.96**	< 0.01	reject	**6.57**	< 0.01	reject
	Clang	**128.15**	< 0.01	reject	**4.75**	< 0.01	reject
	CompCert	**1.74**	0.15	accept	**0.67**	0.62	accept
Cruise Controller	GCC	**126.86**	< 0.01	reject	**30.37**	< 0.01	reject
	Clang	**204.02**	< 0.01	reject	**52.92**	< 0.01	reject
	CompCert	1.81	0.13	accept	6.43	< 0.01	reject

TABLE IV: Comparison between Flag-Use OBC and OBC

Case Example	Compiler	Average Difference	
		Mean	SD
Microwave (C)	GCC	**+0.14**	**-0.18**
	Clang	**+0.64**	+0.05
	CompCert	**+1.08**	**-0.16**
Microwave	GCC	**+6.13**	**-0.51**
	Clang	**+16.21**	**-3.42**
	CompCert	**+2.98**	**-0.75**
Infusion Pump	GCC	**+2.26**	**-0.18**
	Clang	**+2.47**	+0.20
	CompCert	**+0.66**	+0.07
Docking Approach	GCC	**+0.92**	**-0.14**
	Clang	**+4.08**	**-0.65**
	CompCert	**+0.28**	**-0.03**
Cruise Controller	GCC	**+0.78**	**-0.14**
	Clang	**+2.47**	**-0.37**
	CompCert	**+0.45**	**-0.04**

suggest that Flag-Use OBC is more effective in fault-finding than OBC. A lower standard deviation is desired on the other hand, for it shows that Flag-Use OBC is more robust to random sampling of test cases. Based on the desired characteristics, we highlighted with bold-face font the cases where Flag-Use OBC outperformed OBC.

In most cases, Flag-Use OBC outperformed OBC both in terms of higher fault-finding effectiveness and lower standard deviation. The gain in effectiveness was more significant with a higher optimization than with no optimization; in the case of *Microwave* example, for instance, the variance of mutation scores among optimization levels became much less significant compared to the one of OBC. As summarized in the last two columns, the mutation score increased on average in all cases, and the average standard deviation decreased except for a single case, showing that Flag-Use OBC is more effective that OBC both in fault-finding and robustness to compiler configuration.

C. RQ3: Object-level Criteria vs. MC/DC

To answer our final research question, we compared the efficacy of OBC and Flag-Use OBC to the efficacy of MC/DC, which is one of the most rigorous source-level coverage criterion required for Level-A critical software by DO-178B [6]. While MC/DC does not depend on the compiler configuration, object-level coverage criteria does, which makes it difficult to compare them directly. In this comparison, we simply treated all the datasets from different compiler configurations as one, to account for the variations that can be introduced by compilation. The mean and standard deviation for each criterion is presented in Table V per each case example.

TABLE V: Comparison with MC/DC

Case Example	MC/DC		Flag-Use OBC		OBC	
	mean	SD	mean	SD	mean	SD
Microwave (C)	64.95	2.49	63.57	3.35	63.19	3.60
Microwave	65.59	3.11	54.67	5.37	45.38	10.91
Infusion Pump	15.51	1.99	13.17	1.95	11.34	2.61
Docking Approach	17.32	2.36	20.70	1.36	18.93	2.76
Cruise Controller	3.48	1.09	4.11	1.31	3.29	1.79

As discussed in Section V-B, Flag-Use OBC performed better than OBC, approaching closer to the ones of MC/DC. This improvement is mostly due to the gain in higher optimization levels, where Flag-Use OBC was less susceptible to compiler optimization than OBC. In some cases such as *Docking Approach* and *Cruise Controller*, Flag-Use OBC even performed better than MC/DC, scoring 3.38% point and 0.63% point higher than MC/DC, respectively. This result suggests that although OBC is theoretically weaker than MC/DC, its empirical efficacy can be stronger than MC/DC. It also suggests that with a more rigorous object-level coverage criterion, one may achieve a high fault-finding efficacy comparable to MC/DC in the source-level.

D. Threats to Validity

External Validity: We have performed our experiments on five case examples of synchronous reactive critical systems, four of which are written in Lustre and the other in C. We believe that the case examples are representative in the safety-critical domain, and therefore, our findings are generalizable to other systems in the same domain.

We have identified the set of conditional instructions in Intel *IA-64* instruction set, and performed the experiments on Intel x86 architecture. The performance of Flag-Use OBC may depend on the instruction set architecture, for each architecture implements different set of conditional instructions. However, we believe that using Flag-Use OBC will exhibit a similar gain irrespective of the architecture, because many modern instruction set architecture implements conditional behavior through flag registers.

Internal Validity: We have used random test generation and coverage-directed test generation to construct the *master suite* in our experiments. The same test suite is used to construct the suites that satisfy object-level criteria, which means that the master suite does not guarantee the highest achievable coverage in the object level. It is possible that using a different test generation technique to achieve the maximum coverage on the object code may yield a different result.

VI. Discussion

A. What affects the effectiveness of OBC?

Section V presented various factors that can affect the fault-finding efficacy of an object-level coverage criterion, including the choice of compiler, compiler optimization, and coverage criterion. This section discusses in more detail the factors that affected the effectiveness of the object-level criteria.

1) The structure of the source program: In our experiment, one of the factor that affected the variance of mutation score among optimizations most significantly is how the original code was written. The sets of figures for *Microwave (C)* and *Microwave* (Figures in the first and second rows of Table II) illustrate such difference where the variance across optimizations was much higher with the Simulink code than with C code. The minimum mutation score also differed greatly; it was as low as 8% with the Simulink version while it was 49% with the C version.

Considering that the two versions are functionally equivalent, this difference is caused soley by the structural difference in the source code where the translated C code contains many simple branches which are easy to optimize. When optimization is applied on machine-translated C code, most of these branches were optimized to other instructions that incur lesser overhead. Decrease in the number of conditional jump leads to a same degree of decrease in the number of OBC coverage obligation to fulfill, which then compromise the quality or the test suite that OBC requires, eventually plummeting the effectiveness of OBC.

However, we cannot draw a definite conclusion that manually written C code is less susceptible to the compiler configuration, for a lack of extensive empirical study. Understanding such relations requires a thorough study on its own.

2) The number of conditional jumps: To further study the effect of the number of conditional jumps on mutation score, we counted the number of conditional jumps in each object code compiled with different compiler and optimizations. We illustrated the number of conditional jumps for each case example in Figure 4.

The most noticeable trait from the graphs in Figure 4 is a significant decrease in the number of conditional jumps from `-O0` to `-O1` for the case examples written originally in Simulink. The decrease was the most significant when the code was compiled Clang, followed by GCC, while the number stayed almost constant when compiled with CompCert. This trend is consistent with our understanding of the three compiles. CompCert did not optimize as much as others because it is a certified compiler where all of its optimization passes has to be proven for correctness. Clang optimized the jumps more aggressively than GCC possibly due to its more

The trends of decreasing conditional jumps with higher optimizations are also consistent with the drops in the mutation scores as described in Table **??**. Microwave, for instance, showed a huge drop in median mutation score from 57% with `-O0` and 45% with `-O1` when compiled with GCC (Figure (2, 2) in Table II which is consistent with the trend seen in Figure 4a. The difference in median mutation scores between `-O0` and `-O1` was even greater when compiled with Clang, dropping from 53% to a mere 35%, which is also consistent with the trend in Figure 4a where the decrease was greater when compiled with Clang than with GCC.

The number of conditional jumps alone, however, does not explain all the cases. It does not explain Microwave C example, for instance, where the number of jumps is almost constant across optimizations and among compilers. The mutation score still fluctuated, although in a smaller magnitude, across compiler optimizations. In some cases, the median mutation score even increased from `-O0` to `-O1` (Figure (1, 6) in Table II) despite the number of conditional jumps did not change (Figure 4b).

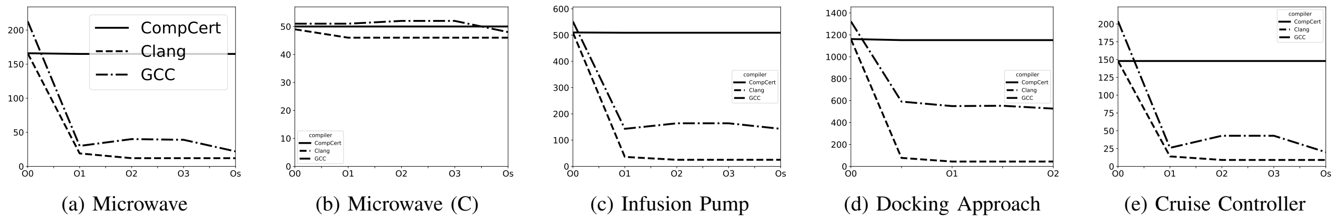
3) Conclusion: Although we have explained several factors that can affect the variance of fault-finding effectiveness to some extent, the list is not complete. Many factors play roles in introducing structural difference in the object code level, including the structure of the source code, compiler, and optimizer, each of which adds another layer of complexity on top of each other.

B. What contributes to the improvement of Flag-Use OBC?

To understand the cause of such the improvement achieved by Flag-Use OBC which is explained in Section V-B, we analyzed the number of flag-use instructions in each binary, which is illustrated in Figure 5. It is notable that the conditional instructions we included in Flag-Use OBC are prevalent in the object code regardless of the compiler used, and in many cases, the number of those instructions increases with higher optimizations. Our conjecture is that when optimization is enabled, compilers prefer to use flag-use instructions in the places of conditional jumps, unless the conditional constructs can be substituted to cheaper non-conditional instructions. Clang, for instance, generated the smallest number of conditional jumps, (Figure 4) but generated the greatest number of other flag-use instructions when optimization is applied (Figure 5).

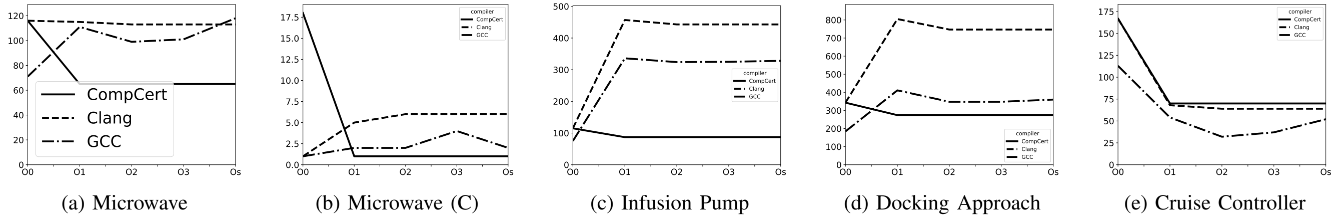
The compilers' preference to generate flag-use instructions is also closely related to the relative gain in mutation score of Flag-Use OBC. For instance, *Docking Approach - Clang* case showed a significant drop in mutation scores with higher optimizations when OBC was used to guide the test construction, while the scores were more stable when Flag-Use OBC was used (Figure (4, 4) and (4, 5) in Table II). The high variance in mutation score with OBC is caused by the stark decrease in the number of conditional jumps as illustrated in Figure 4d. Flag-Use OBC, on the other hand, could also count on the hundreds of other flag-use instructions as illustrated in Figure 5d. As a result, Flag-Use OBC could yield higher mutation scores overall while being more robust to compiler optimizations.

From this observation, we conclude that conditional behavior in the source-level are translated not only to conditional jumps but also to other flag-use instructions. By covering those flag-use instructions, Flag-Use OBC could achieve a higher fault-finding efficacy with a higher robustness to compiler optimizations.



(a) Microwave (b) Microwave (C) (c) Infusion Pump (d) Docking Approach (e) Cruise Controller

Fig. 4: Number of conditional jumps in each object code



(a) Microwave (b) Microwave (C) (c) Infusion Pump (d) Docking Approach (e) Cruise Controller

Fig. 5: Number of flag-use instructions other than conditional jumps in each object code

VII. Related Works

The use of object-level coverage criteria was first proposed by the applicants for DO-178B [6], to be used in the place of source-level coverage criteria as an objective of DO-178B. The issue of using object-level coverage criteria instead of source-level criteria for measuring the adequacy of test suites in safety critical systems was discussed in FAQ 42 of DO-248B [29] (*Can structural coverage be demonstrated by analyzing the object code instead of the source code?*) and CAST paper 17 [7], where FAQ 42 of DO-248B states that: *DO-178B/ED-12B determines the conditions for analysis of the source code for structural coverage, and it does not prevent one from performing analysis directly on the object code.* Although it had not been as widely adopted as MC/DC [30], OBC remains as the primary coverage criterion in the object-code level, especially for the coverage analysis on the object code that is not directly traceable to the source code (Section 6.4.4.2 of DO-178B [29]).

It had once been thought in the past that achieving OBC is sufficient to claim MC/DC [30] when compiler transformation is restricted to short-circuit forms. This equivalence, however, was shown not to hold by Chilenski *et al.* [1], and subsequent in-depth studies followed in an attempt to argue an MC/DC coverage by only performing the object-level coverage analysis. Bordin *et al.* [31] was the first to study this subject, and they characterized the source code construct for which OBC implies MC/DC. Comar *et al.* [32] extended this work to thoroughly understand the relationship between OBC and MC/DC. They formalized the coverage obligations of OBC by casting it to the edge coverage of binary decision diagram (BDD) and proved the conditions under which OBC implies MC/DC. Their theoretical comparison, however, is limited in practice for the following reasons: 1) compiler does not behave the same as they assumed even when the optimization is disabled (see Figure 4), 2) their findings do not hold when compiler optimization is enabled, and 3) a theoretical comparison of coverage criteria cannot alter a probabilistic comparison [33].

Our questions on the effect of code structure on OBC was inspired by Rajan *et al.* [23] who investigated the effect of program structure on the efficacy of MC/DC. Our focus on the effect of compiler optimization is inspired by Hariri *et al.* [34] and Bullseye's statement about the effect of compiler optimization on OBC on their website [8]. This work is also closely related to the empirical comparison performed by Shams and Edwards [35] in which they conclude that OBC is a practical coverage criterion that is comparable to MC/DC. Their study, however, was performed on smaller case examples in a different context of assessing student-written tests, while this work focuses on studying the weakness of OBC and improving itself.

VIII. Conclusion and Future Work

In this paper, we showed how the fault-finding efficacy of OBC is suscpetible to variance in compilers configuration, and proposed a stronger notion of OBC that takes into account conditional instructions in the object code. While this improves the efficacy of branch coverage and is more robust to variations in complier configurations used to produce the object-code, it is still sensitive to the structure of object-code. Recent work on Observable MC/DC [36] effectively addressed a similar issue of the susceptibility of MCDC to code structure by requiring the test cases to propagate the effect of exercising the code-structure to output variables observable to the test oracle. We believe that observability at the object-code level can strengthen Flag-Use OBC and make it robust in the presence of compiler variations.

References

[1] J. J. Chilenski and J. L. Kurtz, "Object-oriented technology verification phase 3 handbook: Structural coverage at the source-code and object-code levels," 2007.

[2] S. Bardin and P. Herrmann, "Structural Testing of Executables," in *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, Feb. 2008, pp. 22–31.

[3] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbe, "Binary-Level Testing of Embedded Programs," in *2013 13th International Conference on Quality Software (QSIC)*. IEEE, Jul. 2013, pp. 11–20.

[4] T. Chen, X.-S. Zhang, X.-L. Ji, C. Zhu, Y. Bai, and Y. Wu, "Test generation for embedded executables via concolic execution in a real environment," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 284–296, 2015.

[5] R. Kersten, S. Person, N. Rungta, and O. Tkachuk, "Improving Coverage of Test Cases Generated by Symbolic PathFinder for Programs with Loops," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–5, Feb. 2015.

[6] L. A. Johnson *et al.*, "Do-178b, software considerations in airborne systems and equipment certification," *Crosstalk, October*, vol. 199, 1998.

[7] CAST, "Structural coverage of object code," Tech. Rep., 2003.

[8] "Code coverage analysis," http://www.bullseye.com/coverage.html#other_object, accessed: Mar 24, 2017.

[9] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[10] I. Jtc, "Sc22/wg14. iso/iec 9899: 2011," *Information technologyProgramming languagesC. http://www. iso. org/iso/iso_catalogue/catalogue_ tc/catalogue_detail. htm*, 2011.

[11] "Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z," https://software.intel.com/en-us/articles/intel-sdm, accessed: April 17, 2017.

[12] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 337–348.

[13] "Qemu: the fast processor emulator," http://www.qemu-project.org/, accessed: Apr 18, 2017.

[14] "bochs: The open source ia-32 emulation project," http://bochs.sourceforge.net/, accessed: Apr 18, 2017.

[15] "Fuzzball: Vine-based binary symbolic execution - bitblaze," https://github.com/bitblaze-fuzzball/fuzzball, accessed: Apr 18, 2017.

[16] "Mathworks stateflow," https://www.mathworks.com/products/stateflow.html, accessed: May 12, 2017.

[17] N. Halbwachs, "Synchronous programming of reactive systems.{T} kluwer academic publishers," *New York*, 1993.

[18] "Verimag lustre v6 tool chain," , accessed: May 12, 2017.

[19] A. Rajan, M. Whalen, M. Staats, and M. P. E. Heimdahl, "Requirements Coverage as an Adequacy Measure for Conformance Testing," in *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, Oct. 2008, pp. 86–104.

[20] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the Danger of Coverage Directed Test Case Generation," in *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer, Berlin, Heidelberg, Mar. 2012, pp. 409–424.

[21] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," in *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*. IEEE, 2001, pp. 83–91.

[22] M. Heimdahl, M. W. Whalen, and A. Rajan, "On MC/DC and implementation structure: An empirical study," ..., pp. 5.B.3–1–5.B.3–13, 2008.

[23] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," *ICSE*, p. 161, 2008.

[24] GCC Team and others, "Gcc, the gnu compiler collection," *dostupno na: https://gcc.gnu.org*, 2013.

[25] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[26] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[27] "Gcc 6.3 manual – options that control optimization," https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Optimize-Options.html#Optimize-Options, accessed: Apr 18, 2017.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065034

[29] I. Rtca, "Final report for clarification of do-178b–software considerations in airborne systems and equipment certification," *Technical Report RTCA/DO-248B*, 2001.

[30] J. J. Chilenski, "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," Apr. 2001.

[31] M. Bordin, C. Comar, T. Gingold, and J. Guitton, "Object and source coverage for critical applications with the couverture open analysis framework," *Embedded Real Time Software and Systems Conference*, 2010.

[32] C. Comar, J. Guitton, and O. Hainque, "Formalization and comparison of MCDC and object branch coverage criteria," *ERTS (Embedded Real Time Systems)*, 2012.

[33] E. J. Weyuker, S. N. Weiss, and D. Hamlet, "Comparison of program testing strategies," in *Proceedings of the symposium on Testing, analysis, and verification*. ACM, 1991, pp. 1–10.

[34] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov, "Evaluating the Effects of Compiler Optimizations on Mutation Testing at the Compiler IR Level," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Sep. 2016, pp. 105–115.

[35] Z. Shams and S. H. Edwards, "Checked Coverage and Object Branch Coverage," in *the 46th ACM Technical Symposium*. New York, New York, USA: ACM Press, 2015, pp. 534–539.

[36] M. W. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats, "Observable modified Condition/Decision coverage." *International Conference on Software Engineering*, 2013.