ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/318295186

Discovering instructions for robust binary-level coverage criteria

Conference Paper · July 2017

DOI: 10.1145/3107091.3107092

CITATION
1

READS
26

5 authors, including:

Vaibhav Sharma
University of Minnesota Twin Cities
3 PUBLICATIONS **1** CITATION

SEE PROFILE

Taejoon Byun
University of Minnesota Twin Cities
6 PUBLICATIONS **20** CITATIONS

SEE PROFILE

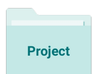
Sanjai Rayadurgam
University of Minnesota Twin Cities
50 PUBLICATIONS **566** CITATIONS

SEE PROFILE

Mats P. E. Heimdahl
University of Minnesota Twin Cities
168 PUBLICATIONS **3,753** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Contract-Based Black-Box Assurance View project

All content following this page was uploaded by Taejoon Byun on 10 July 2017.

The user has requested enhancement of the downloaded file.

Discovering Instructions for Robust Binary-Level Coverage Criteria

Vaibhav Sharma
University of Minnesota
Minneapolis, MN, USA
vaibhav@umn.edu

Taejoon Byun
University of Minnesota
Minneapolis, MN, USA
taejoon@umn.edu

Stephen McCamant
University of Minnesota
Minneapolis, MN, USA
mccamant@cs.umn.edu

Sanjai Rayadurgam
University of Minnesota
Minneapolis, MN, USA
rsanjai@cs.umn.edu

Mats P.E. Heimdahl
University of Minnesota
Minneapolis, MN, USA
heimdahl@cs.umn.edu

ABSTRACT

Object-Branch Coverage (OBC) is often used to measure effectiveness of test suites, when source code is unavailable. The traditional OBC definition can be made more resilient to variations in compilers and the structure of generated code by creating more robust definitions. However finding which instructions should be included in each new definition is laborious, error-prone, and architecture-dependent. We automate the discovery of instructions to be included for an improved OBC definition on the X86 and ARM architectures. We discover all possible valid instructions by symbolically executing instruction decoders for X86 and ARM instructions. For each discovered instruction, we translate it to Vine IR, and check if the Vine IR translation satisfies the OBC definition. We verify the correctness of our tool by comparing its output with the X86 and ARM architecture manuals. Our automated instruction classification facilitates development of more robust OBC definitions with better bug-finding ability and lesser sensitivity to compiler variations.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

instruction classification, object branch coverage

ACM Reference format:
Vaibhav Sharma, Taejoon Byun, Stephen McCamant, Sanjai Rayadurgam, and Mats P.E. Heimdahl. 2017. Discovering Instructions for Robust Binary-Level Coverage Criteria. In *Proceedings of 2017 ACM International Workshop on Testing Embedded and Cyber-Physical Systems, Santa Barbara, CA, USA, July 13, 2017 (TECPS'17),* 4 pages.
https://doi.org/10.1145/3107091.3107092

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
TECPS'17, July 13, 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5112-6/17/07…$15.00
https://doi.org/10.1145/3107091.3107092

1 INTRODUCTION

Object-Branch Coverage (OBC) is a binary-level structural coverage criterion that mandates both sides of conditional branch instructions be exercised by a test suite, when measuring its coverage [2, 3]. OBC has many advantages over its source-level counterparts: it is programming language-independent, and it can be measured in the absence of source code. OBC can be measured non-intrusively and with low execution overhead, for example, using Intel Real Time Instruction Trace [13], or using the Embedded Trace Macrocell architecture [19] on ARM processors. These advantages make OBC desirable for replacing source-level coverage criteria such as branch coverage and modified condition/decision coverage (MC/DC). Further, criteria like OBC are essential for assessing test coverage of object-code that is not directly traceable to source code, which is mandated by standards such as DO-178C [28] for safety-critical avionics software.

Listing 1: C code for `ctermid` function in eglibc-2.19

```
1  char * ctermid (char *s) {
2    static char name[L_ctermid];
3    if (s == NULL)
4      s = name;
5    return strcpy (s, "/dev/tty");
6  }
```

Listing 2: `ctermid` compiled with `-O0`

```
1   40052e: mov    %rsp,%rbp
2   400531: mov    %rdi,-0x8(%rbp)
3           /* if (s == NULL) */
4   400535: cmpq   $0x0,-0x8(%rbp)
5   40053a: jne    400544 <myctermid+0x17>
6           /* s = name */
7   40053c: movq   $0x601041,-0x8(%rbp)
8   400543: /* move s to %rax */
9   400544: mov    -0x8(%rbp),%rax
10          /* %rdx = "/dev/tty" */
11  400548: movabs $0x7974742f7665642f,%rdx
12  40054f: /* move "/dev/tty" to *s */
13  400552: mov    %rdx,(%rax)
14          /* null-terminate *s */
15  400555: movb   $0x0,0x8(%rax)
16  400559: pop    %rbp
17  40055a: retq
```

However, traditional OBC remains susceptible to variations in compilers and compiler optimizations [31]. This susceptibility stems from traditional OBC's reliance on only conditional branch instructions. For example, consider the source code for a C function `ctermid`, shown in Listing 1. `ctermid` [10] returns a string

which, when used as a pathname, refers to the controlling terminal for the current process. Listing 2 presents the disassembly of `ctermid` when compiled with the GNU C compiler with optimizations turned off (`-O0`). Listing 2 contains a `jne` instruction at instruction address `0x40053a`. This conditional branch corresponds with the condition on line 3 of Listing 1. Traditional OBC that relies on only conditional branches would measure coverage based on whether both sides of this `jne` instruction are covered. For this unoptimized compilation, the results match source-level branch coverage. However, consider the disassembly shown in Listing 3.

Listing 3: `ctermid` compiled with `-O1`

```
1              /* %rax = s */
2   40052d: mov    %rdi,%rax
3              /* ZF = (s == NULL) */
4   400530: test   %rdi,%rdi
5              /* move name to %edx */
6   400533: mov    $0x601041,%edx
7              /* %rax = (ZF == 1) ? %rdx : %rax */
8   400538: cmove  %rdx,%rax
9              /* %rcx = "/dev/tty" */
10  40053c: movabs $0x7974742f7665642f,%rcx
11  400543: /* *%rax = "/dev/tty" */
12  400546: mov    %rcx,(%rax)
13             /* null-terminate *%rax */
14  400549: movb   $0x0,0x8(%rax)
15  40054d: retq
```

Listing 3 shows disassembly for `ctermid` when compiled with the GNU C compiler with the `-O1` option. It should be noted that there are no conditional branch instructions in Listing 3. In the `-O1` case, the compiler is able to translate the source-level condition on line 3 of Listing 1 using a `cmove` instruction. Conditional move instructions (commonly referred to as `CMOVcc`) [15] check the state of one or more status flags in the `EFLAGS` register and perform the move operation if the flags are in the specified state. When `ctermid` is compiled with `-O1`, traditional OBC fails to cover the source-level condition, manifested using the `cmove` instruction at the binary level. This lack of coverage provides motivation for updating the definition of traditional OBC.

One way to improve the definition of OBC is to expand the set of instructions that need to be covered. This can be done by going through instruction set architecture manuals. However there are a wide variety of instruction set architectures and variants in use, especially in embedded systems, and instruction sets are large. For instance, the Intel architecture manual is 4700 pages long and the ARM architecture manual is more than 6300 pages long. For every improved OBC definition, manually scanning the architecture manuals to find the set of instructions that satisfy the improved OBC definition would require hundreds of person-hours. It also increases the chance of human errors, either including unnecessary instructions in the updated OBC definition, or missing required instructions. These reasons motivate an automated approach to identifying instructions that satisfy a given OBC definition. A starting point might be to collect instructions from a sample of binaries compiled for a target architecture, but this could miss less common instructions that happened not to appear in the sample. To ensure the entire instruction space is examined, we also want to automate the enumeration of instructions. We present such a principled exploration of X86 and ARM instructions, followed by classification of each discovered valid instruction in Section 2. Section 3 describes related literature, and Section 4 concludes.

2 ENUMERATION AND CLASSIFICATION

As shown in Section 1, 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. Hence, more robust definitions of OBC are needed. Different definitions of OBC can be created based on the kind of coverage of the machine state required. For example, one definition of OBC could include all instructions that read from the `EFLAGS` register. Another definition could include all instructions that write to the `EFLAGS` register. Choosing a more robust OBC definition is a separate research problem that has recently been investigated by Byun et al. [4]. For our evaluation, we use Byun et al.'s definition named *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.

Flag-Use Coverage includes predicated instructions such as `cmovne`, `setne`, and conditional branch instructions like `jne`. We identified X86 and ARM flag-use instructions in three steps. First, we obtained a list of instruction byte sequences for every valid X86 and ARM 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 verified the classification output against the Intel Architecture manual [15], and the ARM manual [1], and obtained a list of instruction mnemonics along with the source of each instruction's conditional behavior.

2.1 Identifying X86 Flag-Use Instructions

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. [21]. 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 fidelity emulator (QEMU [26]) with a high fidelity emulator (Bochs [17]). 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 [8], 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` and `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 set up 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.

Instruction Classification. We independently classified each byte sequence using an instruction decoder using XED [16] and using FuzzBALL and merged the two lists of instruction mnemonics into one by consulting the Intel manual.

XED-based classification: The X86 Encoder Decoder (XED) is a software library for encoding and decoding X86 32 and 64-bit instructions. It is used by Pin [20] and by several other research and commercial projects. One of the outputs of the XED-based instruction decoder is the set of flags read by an instruction. We used this output to check whether an instruction byte sequence execution led to a read of the `EFLAGS` register. 8841 of the 45311 instruction byte sequences read from the `EFLAGS` register, which resulted in a list of 104 unique instruction mnemonics.

FuzzBALL-based classification: FuzzBALL has the ability to execute a single instruction byte sequence (in a FuzzBALL program named `test_insn`). It decodes and executes a single instruction and can be configured to print a simplified Vine IR [30] translation. We extended FuzzBALL's Vine IR translation to identify Vine IR statements that contain either a conditional jump, or a read from one of the six flag register bits (`OF`, `SF`, `ZF`, `AF`, `PF`, `CF`) in the right-hand side Vine IR expression of an assignment statement, or an ITE (if-then-else) ternary operator. Instruction byte sequences, whose Vine IR translation satisfied our checks, were classified as having conditional behavior. Of the 45311 unique byte sequences, 10106 were classified as having conditional behavior, which gave us a list of 178 unique instruction mnemonics. Our Vine IR-based classification was useful in detecting instructions such as `jcxz`, which jumps to the target address if the `CX` register is set to 0. We ran both classifications on a machine running Ubuntu 14.04 with Intel(R) Core(TM) i7-4790 CPU at 3.60 GHz and 16 GB RAM. Our XED-based classification took about 4 minutes, and our FuzzBALL-based classification took about 20 minutes.

Combination with Intel Manual. We took a union of the two lists of instruction mnemonics obtained from XED-instruction decoding-based classification and Fuzzball's Vine IR-based classification. We examined the description of each mnemonic in the Intel manual [15], checking if any instruction behavior reads the non-system flags from the `EFLAGS` register, or is a conditional branch. Any flag bit other than `CF`, `OF`, `PF`, `SF`, `AF`, `ZF` was considered a system flag. This final combination gave us a list of 104 instruction mnemonics. Instructions were excluded in this final combination if they read from a system flag, or had their Vine IR translation use a ITE (if-then-else) expression, but were not conditional branch instructions. We filtered this list based on which of these instruction mnemonics are most commonly used in X86 binaries. This step gave us instructions belonging to the conditional move (CMOVcc), set-byte-on-condition (SETcc), conditional jumps (Jcc) instruction categories, along with the add with carry (ADC) and subtract with borrow (SBB) instructions. We present the list of commonly used instruction mnemonics in Figure 1. Another set of instructions, which read the direction flag (`DF`) (use of `REPE/REPZ/REPNE/REPNZ` causes read of `RCX` and `ZF`, use of `REP` causes read of `RCX`) is: (1) `cmps`, `cmpsb`, `cmpsw`, `cmpsd`, `cmpsq` (2) `ins`, `insb`, `insw`, `insd` (3) `outs`, `outsb`, `outsw`, `outsd` (4) `lods`, `lodsb`, `lodsw`, `lodsd`, `lodsq` (5) `movs`, `movsb`, `movsw`, `movsd`, `movsq` (6) `scas`, `scasb`, `scasw`, `scasd`, `scasq` (7) `stos`, `stosb`, `stosw`, `stosd`, `stosq`. Because these instructions create a loop with the loop count set in `RCX`, they come close to being classified as flag-use instructions. We further validated instructions that read the `EFLAGS` register with Table A-2 in Appendix A of Volume 1 of the Intel manual [14].

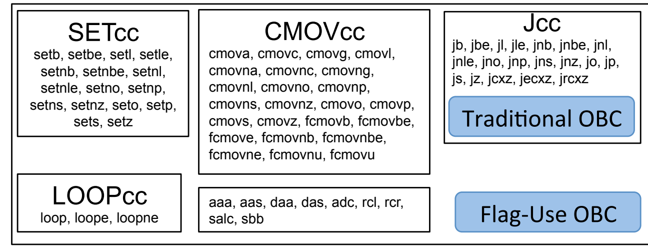


Figure 1: Commonly-used X86 Flag-Use instructions

2.2 Identifying ARM Flag-Use Instructions

Instruction Set Exploration. We discovered flag-use ARM instructions by symbolically executing the instruction decoder in the VEX library of Valgrind [25]. Valgrind is a heavyweight dynamic binary instrumentation framework with support for the ARM platform. VEX is a part of Valgrind's core. It is responsible for dynamic translation of machine code to VEX IR. The first step in this dynamic translation is disassembly of machine instructions to VEX IR opcodes. Each ARM instruction consists of 4 bytes and is given as input to the instruction decoder. We used FuzzBALL to replace this 4-byte input, corresponding to the instruction byte sequence, by 4 symbolic bytes. We symbolically executed the VEX instruction decoder to discover 4306 unique ARM instruction byte sequences.

Instruction Classification. We ran a FuzzBALL-based instruction classification, similar to that done for X86 instructions. We made a small change to make FuzzBALL track reads from the negative flag (`NF`), the zero flag (`ZF`), the carry flag (`CF`), and the overflow flag (`OF`). Running this classification over 4306 ARM instruction byte sequences takes about 3.5 minutes. This classification produces 2612 instruction byte sequences as satisfying the flag-use coverage definition. Refining this list to unique instruction mnemonics gives us 309 instruction mnemonics. Discarding the two-letter suffixes, which indicate condition code values, gives us 70 unique instruction mnemonics. A majority of these allow the instruction execution to be predicated on the condition code. For these instruction mnemonics, the instruction is converted into a *no-op* if the condition code is not satisfied. Of the remaining instruction mnemonics, we find the following mnemonics are not conditional branches but satisfy the flag-use definition: `adc`, `adcs`, `rsc`, `rscs`, `sbc`, `sbcs`. We verified the correctness of our instruction classification by checking the ARM manual.

3 RELATED WORK

The original motivation for achieving binary-level coverage comes from the avionics standard DO-178B [29]. The DO-248B [27] standard discusses substituting source-level coverage (e.g. MC/DC [5]) with object-level coverage. Recovering source-level like conditions from binary code [6] is a related research direction that can help bridge the gap between MC/DC and OBC.

A motivating factor for improving the traditional definition of OBC is its susceptibility to compiler optimizations. Making symbolic execution, mutation testing, and error resilience of software

applications not be susceptible to compiler optimizations are areas of research that have been investigated [7, 11, 24]. Such related techniques can be applied towards creating better OBC definitions.

Instruction set exploration has been the subject of research in the emulator testing community [21–23]. Martignoni et al. [21] describe path exploration lifting, a technique to generate test cases on a high fidelity emulator, and to run them on a low fidelity emulator. Martignoni et al. use FuzzBALL to symbolically execute the instruction decoder of Bochs [17].

Automatic synthesis of symbolic representations of instructions for a processor instruction set is another related area of research. Godefroid et al. [9], and more recently Heule et al. [12], synthesize bit-vector circuits for X86 instructions and discover inconsistencies across X86 processors and errors in the Intel manual [15]. Such inconsistencies motivate the need for our automated instruction classification, assimilated from multiple independent sources.

Our instruction classification implementation can be extended to become a machine code analysis tool that can be retargeted for multiple platforms. Translating instructions from different architectures to a *Universal Assembly Language* [18] would be a starting point for such an extension.

4 CONCLUSION

Object Branch Coverage (OBC) is an important coverage criterion to measure the efficacy of test suites in the absence of source code. With an increasing prevalence of third-party components, and an emphasis on binary-level coverage in aeronatics standards [27], better OBC measurement directly impacts testing. While creating newer and more robust definitions of OBC is important, identifying which instructions should be included in each definition is equally important. Manually reading architecture manuals is laborious, and prone to picking up errors in the manuals [9]. Given a definition of OBC, we present a technique for automatically identifying which X86 and ARM architecture instructions should be included when measuring coverage for the new definition. We present a principled way to discover all possible valid X86 and ARM instruction byte sequences and classify them automatically in a few minutes. Our automated instruction identification promotes development of newer and more robust OBC definitions, which are less sensitive to compiler variations and have better fault-finding efficacy.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 1563920 and 1514444. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors wish to thank Qiuchen Yan for help with enumeration of X86 instructions.

REFERENCES

[1] ARM. 2017. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html. (2017).

[2] Sebastien Bardin, Philippe Baufreton, Nicolas Cornuet, Philippe Herrmann, and Sebastien Labbe. 2013. Binary-Level Testing of Embedded Programs. In *2013 13th International Conference on Quality Software (QSIC)*. IEEE, New York, 11–20.

[3] Sebastien Bardin and Philippe Herrmann. 2008. Structural Testing of Executables. In *2008 International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE, New York, 22–31.

[4] Taejoon Byun, Vaibhav Sharma, Sanjai Rayadurgam, Stephen McCamant, and Mats P. E. Heimdahl. 2017. *Towards Rigorous Object-Code Coverage Criteria*. Technical Report 17-008. Department of Computer Science and Engineering, University of Minnesota.

[5] John J Chilenski. 2001. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. (April 2001).

[6] Adel Djoudi, Sebastien Bardin, and Éric Goubault. 2016. Recovering high-level conditions from binary programs. *FM* (2016).

[7] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. 2015. Studying the influence of standard compiler optimizations on symbolic execution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 205–215.

[8] FuzzBALL 2017. FuzzBALL: Vine-based binary symbolic execution. https://github.com/bitblaze-fuzzball/fuzzball. (2017). Accessed: May 12, 2017.

[9] Patrice Godefroid and Ankur Taly. 2012. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. *SIGPLAN Not.* 47, 6 (June 2012), 441–452.

[10] The Open Group. 2004. ctermid. http://pubs.opengroup.org/onlinepubs/009695399/functions/ctermid.html. (2004). Accessed: June 2, 2017.

[11] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov. 2016. 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)*. 105–115.

[12] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 237–250.

[13] Intel. 2015. Real Time Instruction Trace - Programming Reference. http://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/real-time-instruction-trace-atom-reference.pdf. (2015).

[14] Intel 2017. Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html. (2017). Accessed: May 12, 2017.

[15] Intel. 2017. Intel IA-64 and IA-32 Architectures Software Developer's Manual - Volume 2: Instruction Set Reference, A-Z. https://software.intel.com/en-us/articles/intel-sdm. (2017).

[16] Intel. 2017. Intel XED. https://intelxed.github.io/. (2017). Accessed: May 12, 2017.

[17] Kevin Lawton. 2003. Bochs: The open source IA-32 emulation project. (2003).

[18] Junghee Lim and Thomas Reps. 2013. TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 4 (April 2013), 59 pages.

[19] ARM Limited. 2011. Embedded Trace Macrocell ETMv1.0 to ETMv3.5 Architecture Specification. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_etm_architecture_spec.pdf. (2011).

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (June 2005), 190–200.

[21] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: Hi-Fi tests for Lo-Fi emulators. In *17th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 337–348.

[22] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the 19th Intl. Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/1831708.1831730

[23] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the 18th Intl. Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/1572272.1572303

[24] N. Narayanamurthy, K. Pattabiraman, and M. Ripeanu. 2016. Finding Resilience-Friendly Compiler Optimizations Using Meta-Heuristic Search Techniques. In *2016 12th European Dependable Computing Conference (EDCC)*. 1–12.

[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[26] Peter Maydell et al. QEMU team. 2017. QEMU: the FAST processor emulator. http://www.qemu-project.org/. (2017). Accessed: May 12, 2017.

[27] RTCA. 2001. *Final Report for Clarification of DO-178B–Software Considerations in Airborne Systems and Equipment Certification*. Technical Report. RTCA.

[28] RTCA. 2011. DO-178C, Software considerations in airborne systems and equipment certification. (2011).

[29] Leslie A. Schad. 1998. DO-178B, Software considerations in airborne systems and equipment certification. *CrossTalk* 199 (Oct. 1998), 19.

[30] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, Berlin, 1–25.

[31] BullsEye Testing Technology. 2017. Code Coverage Analysis. http://www.bullseye.com/coverage.html#other_object. (2017). Accessed: May 12, 2017.

View publication stats