# Rapid Taint Assisted Concolic Execution (TACE)

Ridhi Jain
Technology Innovation Institute, UAE
ridhi.jain@tii.ae

Norbert Tihanyi
Technology Innovation Institute, UAE
norbert.tihanyi@tii.ae

Mthandazo Ndhlovu
Technology Innovation Institute, UAE
mthandazo.ndhlovu@tii.ae

Mohamed Amine Ferrag
Technology Innovation Institute, UAE
mohamed.ferrag@tii.ae

Lucas C. Cordeiro
University of Manchester, UK
lucas.cordeiro@manchester.ac.uk

## ABSTRACT

While fuzz testing is a popular choice for testing open-source software, it might not effectively detect bugs in programs that feature many symbols due to the significant increase in exploration of the program executions. Fuzzers can be more effective when they concentrate on a smaller and more relevant set of symbols, focusing specifically on the key executions. We present rapid Taint Assisted Concolic Execution (TACE), which utilizes the concept of taint in symbolic execution to identify all sets of dependent symbols. TACE can evaluate a subset of these sets with a significantly reduced testing effort by concretizing some symbols from selected subsets. The remaining subsets are explored with symbolic values. TACE significantly enhances speed, achieving a 50x constraint-solving time improvement over SymQEMU in binary applications. In our fuzzing campaign, we tested five popular open-source libraries (minizip-ng, TPCDump, GifLib, OpenJpeg, bzip2) and identified a new heap buffer overflow in the latest version of GifLib 5.2.1 with an assigned CVE-2023-48161 number. Under identical conditions and hardware environments, SymCC could not identify the same issue, underscoring TACE's enhanced capability in quickly discovering real-world vulnerabilities.

## CCS CONCEPTS

• **Software and its engineering**; • **Security and privacy → Software security engineering**;

## KEYWORDS

Fuzzing, Symbolic Execution,Taint Assisted Concolic Execution

**ACM Reference Format:**
Ridhi Jain, Norbert Tihanyi, Mthandazo Ndhlovu, Mohamed Amine Ferrag, and Lucas C. Cordeiro. 2024. Rapid Taint Assisted Concolic Execution (TACE). In *Companion Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Symbolic execution (*symbex*) has been around since the 1970s [10, 17], but its application is limited due to high-performance overhead [7]. While symbolic and concolic executions have the potential to address critical program analysis challenges [16, 19, 25], achieving scalable execution faces difficulties, including constraint solving and path explosion [4]. Researchers have suggested strategies to address bottlenecks, such as mitigating path explosion [5, 9] and enhancing constraint-solving [11, 29]. SymCC [23] and SymQEMU [24] port the problem of constraint collection to compile time, thus improving the execution of symbex by not symbolically executing the application. Constraint bloat is a major challenge, making constraint-solving prohibitively expensive [28]. Popular concolic execution solutions symbolize the entire input and emulate all instructions on these symbolized bytes, resulting in many irrelevant constraints. LeanSym [21] (LSym) is an efficient hybrid fuzzer incorporating constraint debloating into symbolic execution through taint flow analysis. Constraint debloating, a technique that involves pruning, simplifying, or optimizing the symbolic constraints, emerges as a crucial approach to mitigate the complexity and overhead associated with symbolic execution. LSym optimizes the concolic component of hybrid fuzzing by conservatively eliminating constraint bloat without sacrificing concolic execution soundness. However, apart from the unavailability of the tool, LSym relies on manual summary creation of relevant syscalls and library functions for function-level tracing.

We present rapid Taint Assisted Concolic Execution(TACE), an automated system that builds on recent advancements in *symbex* technologies, such as SymQEMU and SymCC, but with a notable enhancement in reducing constraint-solving times. We showcase the capabilities of TACE using the most recent versions of five popular libraries in the fuzzer community: minizip-ng, TPCDump, GifLib, OpenJpeg, and bzip2. TACE debloats and optimizes symbolic constraints, facilitating a streamlined symbolic execution process that emphasizes the critical aspects of program behavior. This approach leads to a more feasible and scalable method for uncovering vulnerabilities in real-life scenarios within a reasonable time frame.

Recent techniques, including constraint optimization [18, 20, 21], constraint-based sampling [12], and path pruning [9], target constraint bloat reduction during symbolic execution while still maintaining accurate and complete analysis results. As test applications grow in size and complexity, the number of execution paths rapidly increases, elevating execution time and memory usage. TACE combines taint flow analysis with existing symbex engines (SymCC and SymQEMU) and assists in debloating constraints collected during real execution. This technique reduces the SAT/SMT solver's load to solve the constraints reasonably, thus improving the overall performance of the symbex. This process accelerates symbolic execution, aiding in faster vulnerability detection, input generation, and overall program analysis. From our experience, this approach has proven very effective in today's rapidly evolving environment, where swift and prompt identification of software bugs is essential.

The main original contributions of this work are:

(1) An end-to-end design and implementation of TACE, a tool that leverages taint analysis-based constraint-debloating for efficient symbolic execution;
(2) An extensive evaluation of TACE against the state-of-the-art symbolic execution tool SymCC on five widely-used open-source libraries;
(3) Applying TACE, we discovered a new heap buffer overflow in GifLib 5.2.1, with an assigned CVE-2023-48161 [1] number, a bug that SymCC missed, highlighting TACE's efficiency in quickly detecting real bugs.

In the rest of the paper, Section 2 discusses the related work to TACE. Section 3 presents our proposed tool's design and implementation details. The experimental setup and results are presented in Section 4. Section 5 discusses our findings on a real-world crash. Finally, we conclude this paper in Section 6.

## 2 RELATED WORK

Software testing is pivotal for ensuring the quality of software applications by identifying defects, bugs, and potential vulnerabilities early in the development lifecycle [2, 13]. Symbolic and concolic executions have been popular among testers for their systematic and automated exploration of multiple execution paths within a program. Researchers have proposed improvements to symbolic execution, including KLEE's [5] use of path pruning techniques and MergePoint's [3] alternating between static and dynamic symbolic execution. Other approaches include using compiler optimization recommendations [15] and selected symbolic execution [9]. However, path explosion remains a persistent issue, and pure symbolic execution is limited by the imprecision of static analysis and theorem provers [14]. Concolic execution [6, 14, 26] addresses this by combining symbolic and concrete execution with recent tools such as SymSan [8], SymCC [23], and SymQEMU [24] embedding symbolic processing in the target program. Nagy et al. [22] propose a guided tracing, reducing tracing expenses by filtering and focusing only on test cases that increase coverage, yielding substantial performance improvements in fuzzing. Despite limited tests, the number of constraints can rapidly increase with depth. While path explosion has received extensive research attention, few studies specifically target constraint bloat. LSym [21] utilizes taint analysis to symbolize input bytes influencing the target branch. It employs a hybrid fuzzing technique integrating conservative constraint debloating. However, LSym separately executes symbex and taint flow analysis, resulting in higher overhead. As a solution, TACE groups related constraints with common variables. Further, to eliminate redundancies, TACE iteratively merges the constraint groups with common or dependent variables across two constraint groups.

[1] https://nvd.nist.gov/vuln/detail/CVE-2023-48161

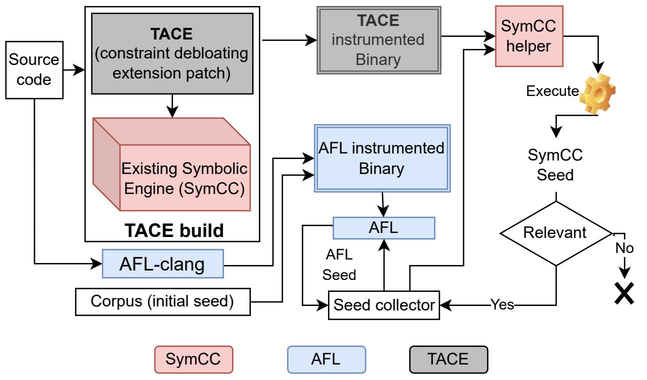

Figure 1: The hybrid fuzzing architecture of TACE on source code with SymCC. A similar architecture is followed by TACE on binaries with SymQEMU.

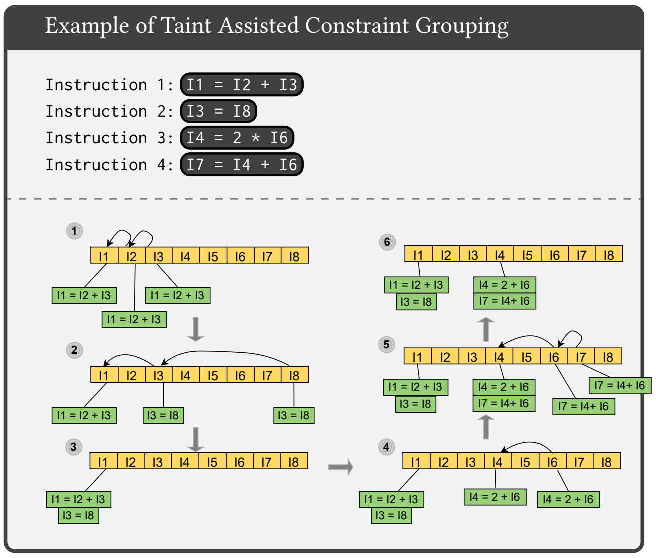
Example of Taint Assisted Constraint Grouping
Instruction 1: I1 = I2 + I3
Instruction 2: I3 = I8
Instruction 3: I4 = 2 * I6
Instruction 4: I7 = I4 + I6
1
I1 I2 I3 I4 I5 I6 I7 I8
I1 = I2 + I3
2
I3 = I8
3
I1 = I2 + I3 I3 = I8
4
I4 = 2 + I6
5
I7 = I4+ I6
6

Figure 2: A four-instruction example to understand TACE.

## 3 DESIGN AND IMPLEMENTATION

TACE is implemented as an extension of the state-of-the-art symbex tools SymCC [23] and SymQEMU [24]. This section discusses the architectural design and implementation of TACE.

### 3.1 The hybrid fuzzing architecture of TACE

Figure 1 illustrates the hybrid fuzzing architecture of TACE on source code with SymCC. The input source project is compiled with TACE and AFL-clang to generate a TACE-instrumented binary and an AFL-instrumented binary, respectively. As TACE uses the underlying architecture of SymCC, it uses *SymCChelper* to generate inputs from the TACE-instrumented binary. AFL [30] generates the initial seed for the AFL-instrumented binary. The AFL-instrumented binary is fuzzed to explore unique crashes and generate new inputs. The seeds generated by *SymCChelper* are only retained if they explore previously unseen paths. The collected seeds are used to fuzz the input program and generate new seeds.

## 3.2 Taint Assisted Constraint Grouping

Consider the four-instruction example in Figure 2 to understand TACE. TACE uses taint labels to group dependent constraints. Taint labels are program variables. Each taint label is one byte, and the maximum number of taint labels for TACE is the total number of bytes in the input program. The taint labels are then associated with the dependent constraints. The constraints are merged, linking them with only one taint label, as illustrated in Figure 2. This approach creates non-overlapping sets of related constraints.

Instruction 1 uses symbols I1, I2, and I3; thus, its associated taint labels are I1, I2, and I3 in ascending order. In the next step, since I1, I2, and I3 interact in the constraint of Instruction 1, the set of constraints associated with I3 is moved to (and attached with) I2, and then the set of constraints related to I2 is moved to I1. The above process is repeated for each instruction as it is encountered.

The systematic handling of constraints based on instruction interactions and taint labels facilitates a precise assessment of the overall taint propagation within the program, contributing to a deeper comprehension of TACE and its implications for security and analysis in program execution.

## 4 EXPERIMENTAL EVALUATION

## 4.1 Experimental setup and Benchmarks

The experiments were conducted on Ubuntu 22.04.1 (5.15.0-67-generic) with 256 GB RAM, AMD Ryzen Threadripper PRO 3955WX 16-Cores CPU. We evaluate TACE against SymCC on five widely used open-source C++ libraries: *GifLib*, *TCPdump*, *minizip*, *Openjpeg*, and *bzip2.* The selected libraries are compiled with `afl-clang` and TACE to generate AFL and TACE instrumented binaries, respectively. AFL assists in seed generation and fuzzing.

## 4.2 Research Questions

- ***RQ1:*** What performance gains are offered by TACE compared to state-of-the-art tools, SymCC and SymQEMU?
- ***RQ2:*** Does TACE report correct and reproducible bugs?

## 4.3 Results

*4.3.1 Constraints Solving Time.* Reducing constraint-solving time significantly expedites bug discovery in actual applications. To exhibit TACE's practical efficiency, we crafted `symex2.c`[2], a small sample code resembling real-world projects with multiple if-then-else branches, making achieving full code coverage quite challenging. We evaluated TACE's constraint debloating capabilities by comparing the time taken by SymQEMU and TACE at varying depth levels for the `symex2.c` program. The results, as shown in Table 1, highlight TACE's substantial improvement in solving symbolic constraints compared to the state-of-the-art tool SymQEMU.

Particularly, TACE's performance exhibits notable improvement as the program's depth increases, mainly due to introducing more variables during deeper analysis, escalating the complexity of constraints for SymQEMU. However, TACE's constraint debloating strategy leads to an *exponential reduction* in constraint-solving time. The data in Table 1 indicates that as the program's depth grows, TACE observes more concrete dependencies while symbolic dependencies

[2]https://github.com/tacetool/TACE/blob/main/tace/symcc/test/symex2.c

**Table 1: SymQEMU vs. TACE: Constraint Solving Time at Various Depths**

| Depth | SymQEMU (Sec) | TACE (Sec) | Symbolic Dep | Concrete Dep | Improvement (x times) |
|---|---|---|---|---|---|
| 1 | 4.28 | 4.30 | 3 | 2 | 0.99 |
| 2 | 7.44 | 5.69 | 4 | 3 | **1.30** |
| 3 | 9.76 | 6.26 | 4 | 4 | **1.55** |
| 4 | 15.02 | 7.84 | 4 | 6 | **1.91** |
| 5 | 26.11 | 9.35 | 4 | 10 | **2.79** |
| 6 | 122.65 | 7.81 | 4 | 18 | **15.70** |
| 7 | 145.50 | 4.38 | 4 | 34 | **33.21** |
| 8 | 280.48 | 5.59 | 4 | 66 | **50.17** |

remain relatively stable. This reduction in symbolic dependencies by TACE significantly decreases solving time and facilitates the generation of pertinent inputs.

*4.3.2 Fuzzing Results.* Fuzzing dynamically explores possible software behaviors as per the inputs provided. The relevance of these inputs directly affects the new paths explored. Code coverage of a fuzz campaign is directly linked to the quality of inputs. Identifying unique and relevant inputs for bigger constraints is much harder than curating the inputs from smaller constraints. We measure the following metrics as indicators of the target exploration quality:

- **Unique Timeouts** are cases where the fuzzed program exceeds a defined time limit for execution;
- **Unique Hangs** refer to instances where the fuzzed program becomes unresponsive or "hangs". The default timeout for all experiments was set to 1 second;
- **Unique Crashes** represent distinct instances where the program being fuzzed crashes;
- **Paths** represent the unique sequences of operations or code executed during a fuzzing session;
- **Edges** ar breakable, e the transitions or branches taken within the program during fuzzing;
- **Cycles** indicate revisiting a previously encountered state.

If persistent with diverse seeds, fuzzers unveil numerous unexplored paths over time. Yet, the real advantage is in an efficient fuzzer that generates highly effective seeds, enabling the *early detection* of crashes and hangs. To serve this objective, we fuzz test each project (*minizip-ng*, *TPCDump*, *GifLib*, *OpenJpeg*, *bzip2*) for 24 hours. Our experiments indicate TACE provides higher coverage than existing state-of-the-art tools, thus uncovering newer bugs in fewer explorations. TACE's intelligent debloating approach generates relevant inputs by modifying only impactful variables.

Table 2 presents our experimental findings for five chosen open-source libraries, contrasting TACE against SymCC. The data outlines unique hangs, timeouts, and crashes encountered during fuzzing. It also includes metrics regarding newly discovered edges, cycles, and explored paths. TACE outperforms SymCC in several key aspects. For instance, in a 24-hour analysis of the *minizip-ng* project, TACE identified a significantly greater number of new edges, totaling 3,631 compared to just five new edges discovered by SymCC. TACE also surpasses SymCC regarding unique hangs and crash detection. For instance, in our experiments, TACE detected 36 hangs in *TCPDump* and 10 hangs in *OpenJpeg*, compared to 14 and 0 hangs detected by SymCC, respectively. Similarly, TACE reported 1201 unique time-outs for *TCPDump* and 894 time-outs for *OpenJpeg* as opposed to 336 and 1 reported by SymCC, respectively, in the same experimental setup. TACE identified seven distinct crashes in GifLib 5.2.1, whereas SymCC reported none. To confirm these findings,

Table 2: Comparision hybrid fuzzing statistics from SymCC and TACE.

| Test Details | | SymCC | | | | | TACE | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Target Project | Time (hh:mm:ss) | #Unique Hangs | #Unique tmouts | #Unique Crashes | #New Edges | #Cycles | #Unique Hangs | #Unique tmouts | #Unique Crashes | #New Edges | #Cycles |
| minizip-ng | 24:00:00 | 0 | 4 | 0 | 5 | 391 | 0 | 203 | 0 | 204 | 3.6k |
| TCPDump | 24:00:00 | 14 | 336 | 0 | 4148 | 223 | 36 | 1201 | 0 | 7 | 286 |
| GifLib | 24:00:00 | 2 | 71 | 0 | 78 | 108k | 13 | 62 | 7 | 72 | 181k |
| OpenJpeg | 24:00:00 | 0 | 1 | 0 | 3 | 89 | 10 | 894 | 0 | 71644 | 104k |
| bzip2 | 24:00:00 | 0 | 2 | 0 | 4 | 76.2 | 0 | 2 | 0 | 4 | 229k |

an exhaustive manual verification is vital. After verification, this information must be communicated to the developers (if valid). The results of this verification and detailed explanations will be presented in the following section.

## 5 EXPLORING CVE-2023-48161: A CASE STUDY

GifLib, a widely used library for handling GIF image files, is crucial in various applications and systems. GifLib is included in the Fedora project, thereby is a default library in both Fedora and Ubuntu Linux distributions. To ensure that GifLib resists various input scenarios and edge cases, TACE generates random and mutated inputs, including malformed GIF files, by combining mutation algorithms with the initial seed corpus. These inputs are systematically fed into GifLib, executing multiple test cases with varied inputs. TACE monitors the execution of each test case and detects crashes or unexpected behaviors, indicating potential vulnerabilities.

### 5.1 Results

After running TACE against GifLib 5.2.1 over 24 hours, the following results were obtained:

(1) **Unique Crashes** As illustrated in Figure 3, TACE can detect crashes in GifLib early in its analysis as it generates more relevant inputs.
(2) **Unique Hangs** In our experiments on GifLib, as shown in Figure 3, the unique hangs grow persistently with time, especially in later inputs, indicating that the early inputs are more useful in detecting crashes.
(3) **Levels** The depth of the test application explored is referred to as level. The saturation in levels indicates that the maximum depth is explored.

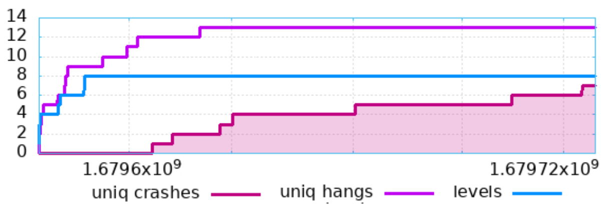

Figure 3: Unique crashes, hangs, and levels in mili-seconds (GifLib fuzzing).

The fuzzing case study with TACE on GifLib 5.2.1 demonstrated its effectiveness in identifying potential vulnerabilities, crashes, and memory leaks.

### 5.2 Validating Results for Correctness

TACE found 7 distinct crashes in GifLib 5.2.1, all stemming from a single heap buffer overflow. The findings were reported to the developers after manual verification via Google's `AddressSanitizer` [27]. The overflow within the dynamic memory allocation of the GifLib program led to a situation where data was written past the allocated memory region in the heap, triggering a critical fault. A similar heap overflow vulnerability in this application was previously documented under the CVE identifier CVE-2022-28506. Fedora implemented a patch for that vulnerability [3], however the issue recently pinpointed by TACE, specifically in the image-saving process at line 321 of the `gif2rgb.c` file, continues to exist. The vulnerability is now officially reported and given the CVE number **CVE-2023-48161**.

Tables 2 and 1 combinedly answer **RQ1**. Table 1 showcases the speed-up offered by TACE compared to pure symbolic execution employed by SymQEMU. Table 2 emphasizes the crash discovery capability of TACE compared to SymCC, which does not discover any in the given 24-hours period. Our experiments show that TACE reports real and reproducible crashes corresponding to **RQ2**.

### 5.3 Reproducibility

We have made TACE accessible to the research community [1]. We offer the necessary resources for compiling TACE from its source code and have also created a Dockerfile to simplify its usage. The repository also contains the `symex2.c` example discussed in Table 1. We provide detailed instructions in the same repository for replicating the CVE-2023-48161 heap buffer overflow, as described in this section. A demo video for TACE is available at: https://www.youtube.com/watch?v=FZpxPNsp_IE.

### 5.4 Threat to validity

Although TACE offers a significant performance gain over existing techniques, it also inherits shortcomings of concolic execution. The approach can detect vulnerabilities and bugs faster than existing approaches, yet it is not a complete solution and, thus, may not report all existing vulnerabilities. In addition, TACE maintains the taint information and, thus, is memory intensive.

## 6 CONCLUSION

TACE is a platform-independent approach that can be applied to any existing tool for performance enhancement. TACE is evaluated as an extension on SymCC and SymQEMU, thereby accepting source files and binaries, respectively. TACE shows promising results in debloating program constraints, surpassing the performance of current methods. Real-world case studies demonstrate TACE's effectiveness in uncovering hidden vulnerabilities and enhancing the robustness of software projects, evidenced by its detection of a new heap buffer overflow, CVE-2023-48161, in the well-tested GifLib project. We envision a future of continual refinement and innovation for our tool and encourage collaboration and community contributions.

[3]https://bodhi.fedoraproject.org/updates/FEDORA-2022-964883b2a5

## REFERENCES

[1] 2023. Taint Assisted Concolic Execution (TACE) GitHub repository. https://github.com/tacetool/TACE.

[2] Kaled M. Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C. Cordeiro. 2021. FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs. In *15th International Conference on Tests and Proofs (TAP) (LNCS, Vol. 12740)*, Frédéric Loulergue and Franz Wotawa (Eds.). Springer, 85–105.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *ICSE*. 1083–1094.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX* (San Diego, California) *(OSDI)*. 209–224.

[6] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. *ACM TISSEC* 12, 2 (2008), 1–38.

[7] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Symposium on Principles of Programming Languagesthree decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[8] Ju Chen, WookHyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2531–2548. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[10] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering* 3 (1976), 215–222.

[11] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. 2015. SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In *SAT 2015*. Springer, 360–368.

[12] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. 2018. SMTSampler: Efficient stimulus generation from complex SMT constraints. In *2018 IEEE/ACM ICCAD*. 1–8.

[13] Mikhail R. Gadelha, Rafael S. Menezes, and Lucas C. Cordeiro. 2021. ESBMC 6.1: automated test case generation using bounded model checking. *Int. J. Softw. Tools Technol. Transf.* 23, 6 (2021), 857–861.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *PLDI*. 213–223.

[15] Wei-jiang Hong, Yi-jun Liu, Zhen-bang Chen, Wei Dong, and Ji Wang. 2020. Modified condition/decision coverage (MC/DC) oriented compiler optimization for symbolic execution. *Frontiers of Information Technology & Electronic Engineering* 21, 9 (2020), 1267–1284.

[16] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *9th International Conference, TACAS, ETAPS*. Springer, 553–568.

[17] JC King. 1976. Symbolic Execution and Program Testing, communications de l'ACM, vol. 19 n. 7. *July* 10 (1976), 360248–360252.

[18] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. 2018. Learning SMT (LRA) constraints using SMT solvers. In *IJCAI*. 2333–2340.

[19] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. 2013. Software vulnerability detection using backward trace analysis and symbolic execution. In *2013 International Conference on Availability, Reliability and Security*. IEEE, 446–454.

[20] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic Optimization with SMT Solvers. In *POPL* (San Diego, California, USA). ACM, 607–618. https://doi.org/10.1145/2535838.2535857

[21] Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2021. LeanSym: Efficient hybrid fuzzing through conservative constraint debloating. In *RAID*. 62–77.

[22] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE SP*. 787–802.

[23] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *USENIX*. 181–198.

[24] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS*. Internet Society.

[25] Richard Rutledge and Alessandro Orso. 2022. Automating Differential Testing with Overapproximate Symbolic Execution. In *IEEE ICST*. 256–266.

[26] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools: (Tool Paper). In *CAV 2006*. Springer, 419–423.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[28] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. 2015. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *ISSTA*. ACM, 199–210.

[29] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. 2009. A concurrent portfolio approach to SMT solving. In *CAV*. Springer, 715–720.

[30] Michał Zalewski. [n. d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. [Online; accessed 1 Sep. 2022].