FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers

Haoxin Tu*
Singapore Management University
Singapore
haoxintu.2020@phdcs.smu.edu.sg

Lingxiao Jiang, Xuhua Ding
Singapore Management University
Singapore
{lxjiang,xhding}@smu.edu.sg

He Jiang
School of Software, Dalian University of Technology, Dalian, China
jianghe@dlut.edu.cn

ABSTRACT

Symbolic execution (SE) has been widely adopted for automatic program analysis and software testing. Many SE engines (e.g., KLEE or Angr) need to interpret certain Intermediate Representations (IR) of code during execution, which may be slow and costly. Although a plurality of studies is proposed to accelerate SE, few of them consider optimizing the internal interpretation operations. In this paper, we propose FastKLEE, a faster SE engine that aims to speed up execution via reducing redundant bound checking of type-safe pointers during IR code interpretation. Our two key insights are: (1) the number of interpreted instructions can be tremendous and reducing the interpretation overheads of the extensively interpreted ones (e.g., read/write) could potentially accelerate the execution; (2) a large portion of the pointers in C programs can be statically verified to be type-safe, but existing SE engines treat all the pointers equally, which indicates that those engines may slow down the execution due to unnecessary bound checking. Specifically, in FastKLEE, a type inference system is first leveraged to classify pointer types (i.e., *safe* or *unsafe*) for the most frequently interpreted read/write instructions. Then, a customized memory operation is designed to perform bound checking for only the *unsafe* pointers and omit redundant checking on *safe* pointers. We implement FastKLEE on top of the well-known SE engine KLEE and combined it with the notable type inference system CCured. Evaluation results demonstrate that FastKLEE is able to reduce by up to 9.1% (5.6% on average) as the state-of-the-art approach KLEE in terms of the time to explore the same number (i.e., 10k) of execution paths. FastKLEE is open-sourced at https://github.com/haoxintu/FastKLEE. A video demo of FastKLEE is available at https://youtu.be/fjV_a3kt-mo.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Symbolic execution**; **Performance**.

KEYWORDS

Software testing, symbolic execution, performance, type inference

*Also affiliated with the School of Software, Dalian University of Technology.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558919

ACM Reference Format:
Haoxin Tu, Lingxiao Jiang, Xuhua Ding, and He Jiang. 2022. FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558919

1 INTRODUCTION

Symbolic execution (SE) is a prominent technique that has been applied in many areas, such as software engineering [2, 5, 12], programming language [16, 17, 22], and security [1, 6, 8, 30]. The key idea of SE is to simulate program executions by using symbolic values for inputs and then each execution path will be encoded as path constraints during execution. A constraint solver (e.g., STP [25] or Z3 [31]) later is used to determine the feasibility of each path constraint, and the solved paths are further explored. Among all the existing SE engines, the IR-based ones (e.g, KLEE [5], S2E [8], or Angr [24]), whose execution is conducted by interpreting the Intermediate Representations (IR) of the target program under test, are prevalent and widely used. Typically, traditional IR-based SE first transforms the program (either source code or binary) under analysis into IR, and then it interprets the IR to execute the program, which follows the design in routes ① → ② → ③ in Figure 1. Such a design can have manifest benefits from the implementation perspective. Instead of interpreting numerous instruction sets of popular CPU architectures, IRs typically represent program behavior at a high level with fewer instructions, thus making it easier to implement an instruction interpreter for architecture-independent instruction sets than manipulating complex instructions directly.

Although the design of IR-based SE is convenient in terms of implementation, existing studies [22, 23] point out that such a design carries out significant performance downsides and slows down the execution. To mitigate such a problem, two major flavors of studies are devoted to accelerating SE. Based on the fact that IR code can be transformed by aggressive optimizations, the first promote directions are dedicated to either selecting existing compiler optimizations [7, 11] that could help accelerate SE or designing a stand-alone customized optimization [29] for program verification (e.g., symbolic execution). Apart from the IR optimization side, other solutions such as reducing the number of paths to be explored [3, 27] or optimizing the constraint solving [15, 21] are considered for speeding up the execution process. However, most of the existing studies neglect the internal interpretation downside of IR-based SE in terms of performance. That is, IR code interpretation alone could slow down the performance of SE engines.

In this study, we propose FastKLEE, a tool that aims to support faster SE by reducing the interpretation overheads of redundant

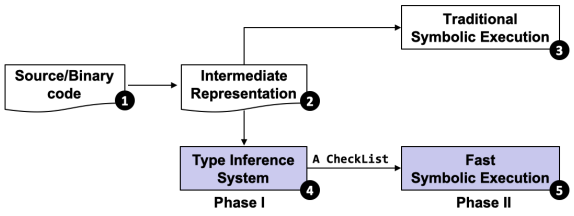

Figure 1: Overview of IR-based *Traditional* SE and *Fast* SE

bound checking of type-safe pointers. We design FastKLEE based on the following two intuitions. First, the number of interpreted instructions tends to be stupendous (usually billions only in one hour's run), and reducing overhead for the most frequently interpreted ones (i.e., read/write) could potentially accelerate the execution. Second, type-safe is an important property in the program for preventing certain errors such as memory out-of-bound assessing. As proven by prior studies [14, 19, 20], a large portion of pointers in C programs to be read/write can be statically verified to be type-safe. However, most of the existing IR-based SE engines treat all the pointers (memory addresses) equally and perform the bound checking for every pointer when interpreting read/write instruction. Thus, a plethora number of bound checking performed during the interpretation is unnecessary, which may induce performance downsides. Based on the above intuitions, we propose FastKLEE to support faster SE. Specifically, in FastKLEE, a type inference system is leveraged to perform type inference before executing the SE engine (which follows the design in routes ① → ② → ④ → ⑤ in Figure 1, where the latter two are designed for faster SE). The type inference system (i.e., ④) is first used to statically verify pointers to be *safe* or *unsafe* and produce a checking list `CheckList` for *unsafe* pointers. Then, during the execution in ⑤, a customized memory operation is designed to perform bound checking only for *unsafe* pointers (stored in `CheckList`), while *safe* pointers are no longer needed to perform any redundant checking. In this way, FastKLEE could reduce the interpretation overheads of redundant bound checking of type-safe pointers for faster SE.

We implement FastKLEE on top of the well-known SE engine KLEE [5] and the type inference system CCured [20]. To demonstrate the effectiveness of FastKLEE, we compare it with the state-of-the-art approach KLEE over widely-adopted GNU `Coreutils` datasets. The evaluation results demonstrate that FastKLEE is able to reduce by up to 9.1% (5.6% on average) as KLEE in terms of the time to explore the same number (i.e., 10k) of execution paths.

In summary, this paper makes the following contributions:

- To our best knowledge, FastKLEE is the first SE engine that aims to accelerate IR-based SE by reducing interpretation overheads.
- We leverage a type inference system to classify pointers and design a customized memory operation in the SE engine to avoid redundant checking of type-safe pointers, thus facilitating the reduction of interpretation overheads for IR-based SE.
- We open-source the tool FastKLEE and demonstrate its usability and effectiveness. We also discuss several important implications of FastKLEE, such as facilitating valuable path exploration.

Organizations. Section 2 describes the workflow of FastKLEE for end-users. Section 3 gives the most related works to our approach. Sections 4 and 5 describe the design and implementation of FastKLEE. Section 6 presents the evaluation results. Section 7 discusses potential implications, threats to the validity, and limitations of our approach. Section 8 concludes this paper with future work.

2 USAGE EXAMPLE

Users can execute "./setup.sh" in the code repository to set up FastKLEE. To prepare the test programs under test, it is recommendable for users to follow the official instruction [9] to get the LLVM bitcode files of test programs to be analyzed (e.g., `cat.bc` utility in GNU `Coreutils`). After setting up the tool and the test program, the following two major phases are considered to use FastKLEE through a command-line interface.

2.1 Phase I: perform type inference

In the first phase, the target test program `cat-linked.bc` under testing will be first instrumented to be a new bitcode file named `cat-linked.bc` by invoking LLVM tool-chain `llvm-link`. Then, the `ccured` pass will be applied on the `cat-linked.bc` by invoking the tool `opt`. After the type inference of interpreted pointers, a text file `cat-checklist.txt` will be produced in the current folder, which will be used later in the next phase.

```
$llvm-link cat.bc nescheclib.bc -o cat-linked.bc
$opt -load libccured.so -nescheck -stats -time-passes < cat-linked.
     bc >& /dev/null
```

2.2 Phase II: conduct faster symbolic execution

In the second phase, users can utilize the same command line with the original KLEE to perform SE upon the test program. Specifically, users may follow the official document [10] to opt for the applicable `options`. During the running of FastKLEE, the file `cat-checklist.txt` will be loaded first and then will be used to guide the customized memory operation in FastKLEE.

```
$fastklee [options] ./cat.bc --sym-args 0 1 10 --sym-args 0 2 2 --
     sym-files 1 8 --sym-stdout
```

3 RELATED WORK

The most related work to us can be broadly divided into two categories: compiler optimization-based and compiler optimization agnostic. In the former, Dong et.al [11] study the influence of standard compiler optimizations on SE, and Chen et.al [7] further leverage machine-learning-based compiler optimization tuning to select a set of optimizations to accelerate SE. Jonas et.al [29] later design a stand-alone optimization for optimizing programs for fast verification, which includes accelerating SE. In the latter, different approaches are proposed to reduce the number of paths to be explored [5, 17, 18, 27] or optimize the constraint solving [5, 21, 28] in SE, thus speeding up the execution process.

Unlike the existing approaches, our goal is to make IR-based SE more efficient by reducing the internal interpretation overheads, i.e., we aim to reduce redundant bound checking of type-safe pointers during IR code interpretation. It is worth noting that our approach can be complementary to existing approaches and further boost faster SE by combing them with our proposed approach.

Algorithm 1: Type Inference System in FastKLEE

```
   Input: a IR code file bc
   Output: a checking list of unsafe pointers CheckList
1  Function typeInferenceFunc(bc):
2      CheckList ⟵ ∅ // initialize a checking list
3      Instruction *i = processInst(bc)
4      switch i->getOpcode() do
5          ... // other instructions
6          case Instruction::Load do
7              ptr = classifyPointer(i)
8              if ptr.type != SAFE then
9                  key = generateKey(ptr, i)
10                 CheckList.append(key)
11         case Instruction::Store do
12             ptr = classifyPointer(i)
13             if ptr.type != SAFE then
14                 key = generateKey(ptr, i)
15                 CheckList.append(key)
16         ... // other instructions
17     return CheckList
```

Algorithm 2: Customized Memory Operation in FastKLEE

```
   Input: a checking list of unsafe pointers CheckList
1  Function MemoryOperation(state, ki, CheckList):
2      inBound ⟵ ∅ // initialize a Boolean value
3      key = generateKey(ki)
4      if CheckList.find(key) then
5          inBound = normalChecking(state, ki)
6      else
7          inBound = 1
8      ... // the following handling
```

4 APPROACH

In this section, we present the detailed design of FastKLEE. As shown in routes ① → ② → ④ → ⑤ in Figure 1, after obtaining the IR code, FastKLEE first leverages a type inference system in ④ to classify different kinds of pointers and store the *unsafe* pointers in a checking list (i.e., `CheckList`). Then, the `CheckList` is passed to ⑤ and will be used in the customized memory operation in FastKLEE. More specifically, if a pointer under read/write operation is in the checking list, a normal bound checking is conducted. Otherwise, FastKLEE omits the bound checking and continues to the interpretation. In short, the omitted portion of bound checking during memory operation is the weapon inside FastKLEE to make faster SE. Next, we describe the details of the type inference system and customized memory operation designed in FastKLEE.

4.1 Type Inference System

The type inference system in FastKLEE is designed for classifying pointer types. Typically pointer types in a type inference system are in the following three forms: (1) ***SAFE*** pointer can only be null and only needs a null-pointer check at runtime, (2) ***SEQ*** pointer can be null, be interpreted as an integer, or be manipulated via pointer arithmetic. At runtime, it needs a null-pointer and bounds check, and (3) ***WILD*** pointer cannot be statically typed and it needs null-pointer, bounds, and dynamic type checks at runtime.

Since most IR-based SE engines lack runtime information during execution, i.e., programs are not executed like native runs, we categorize pointer types only into *safe* and *unsafe* (i.e., SEQ and WILD) in FastKLEE. Specifically, the *safe* pointers can be statically verified to be type-safe so they do not need bound checking during interpretation, while only the *unsafe* pointers are needed to be bound-checked. Therefore, the functionality of the type inference system in FastKLEE is, given an IR code of the program under test, it records the *unsafe* pointers when the pointer is performed in the read/write operations. When all the instructions are inferred, a checking list `CheckList` that stores all the *unsafe* pointers is returned. Such a checking list will be the guidance for reducing redundant bound checking of type-safe pointers in FastKLEE.

Algorithm 1 presents the detail of the type inference system introduced in FastKLEE. The function `typeInferenceFunc` is responsible for classifying different types of pointers. It takes an IR code file `bc` as input and outputs a checking list `CheckList`. Inside the function, it first initializes the `CheckList` in Line 2 and accordingly processes instructions in the `bc` file in Line 2. Then, when encountering a read (Line 6) or write (Line 11) instruction, the pointer (i.e., the memory address under read/write) is classified by invoking the function `classifyPointer` (in Line 7 or 12). Later, an *if-branch* is performed to check whether the `ptr` under handling is an *unsafe* pointer (in Line 8 or 13). If the answer is yes, a `key` that represents a unique pointer is generated by calling the function `generateKey` and will be stored in the `CheckList` later (in Lines 9-10 or Lines 14-15). Finally, the checking list is returned in Line 17 and will be used in the customized memory operation in FastKLEE.

4.2 Customized Memory Operation

The purpose of the customized memory operation in FastKLEE is to take the output (i.e., `CheckList`) from the type inference system and use it to guide the customized memory operation during interpretation. Algorithm 2 describes the details. First, a Boolean variable `inBound` is initialized in Line 2 and the `key` is retrieved by calling the function `generateKey` in Line 3 when a read/write instructions are interpreted. Then, a checking of whether the `key` is in the `CheckList` is performed in Line 4 to process either normal checking in traditional execution in Line 5 if the checking returns *true* or assignment of `inBound` to 1 in Line 7 if the checking returns *false*. Later, the normal execution continues after Line 8.

By equipping the type inference system and the customized memory operation, FastKLEE is capable of reducing the interpretation overheads of redundant bound checking of type-safe pointers.

5 IMPLEMENTATION

We implemented FastKLEE on top of KLEE (version 2.1) and combined it with the well-known CCured type inference system [20]. Specifically, for the CCured system, we implement it on top of DataGrad [13]. In particular, due to the unmapped IR information between analysis and original IR (refers to an issue[1] for more

[1] https://github.com/Lightninghkm/DataGuard/issues/2

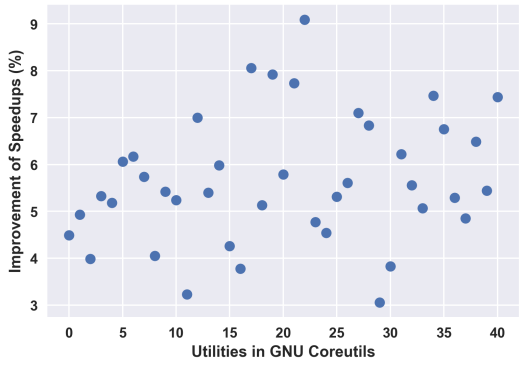

Figure 2: Scatter plot of the improvement in speedups

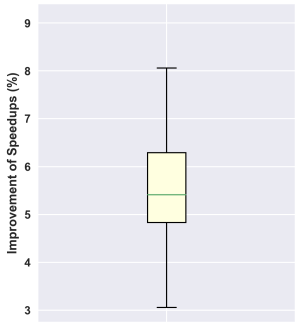

Figure 4: Box plot of the improvement in speedups

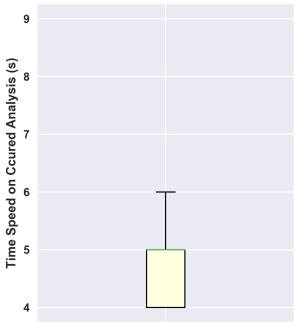

Figure 5: Box plot of the time spent on type inference

details), we first use the debug information (i.e., file name, function name, the line number of instruction, and column number of instruction) inside the instruction to represent a unique `key` (implemented in Algorithm 1). Then, we reuse the APIs from DataGrad to record/store the *unsafe* pointers into a checking list `CheckList`. For the implementation of the customized memory operation in FastKLEE, we modified KLEE's memory operation API to leverage the information (i.e., `CheckList`) from type inference analysis, and decide whether the bound checking of a pointer is needed.

6 EVALUATION

This section presents our evaluation setup and results. Specifically, we demonstrate the benefits of FastKLEE in terms of the time spent on exploring the same number (i.e., 10k) of execution paths.

6.1 Experimental setup

Benchmark. We evaluate FastKLEE on the widely used GNU `Coreutils` (version 9.0) benchmarks. Specifically, we select 40 programs in it, and the excluded utilities can be categorized into the following types: (1) cause non-deterministic behaviors (e.g., `kill`, `ptx`, and `yes`), following existing studies [12, 18], (2) exit early due to the unsupported assembly code or external function call, and (3) can not successfully explore 10k execution paths in 2 hours (i.e., the timeout to run each test program we set in the experiment).

Approach under comparison. We adopt the notable SE engine KLEE as our baseline, as we built on top of it.

Running settings. We followed prior work [5, 18] to set symbolic inputs for GNU `Coreutils` programs. Besides, we use *Breadth First Search* to deterministically guide the path exploration in KLEE and FastKLEE. The experiments are conducted on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz × 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

6.2 Evaluation Results

To demonstrate the effectiveness of FastKLEE, we run FastKLEE against KLEE in terms of the time spent on exploring the same number of explored paths over GNU `Coreutils`. We set a timeout of 2 hours to run each program and count the time spent on exploring certain execution paths. Specifically, we follow the formula below to calculate the speedups:

$$\frac{T_{baseline} - T_A}{T_{baseline}} \times 100\%$$

where $T_{baseline}$ represents the time spent by the baseline KLEE on exploring 10k paths and T_A describes the time spent by our proposed FastKLEE on exploring the same number of paths.

The scatter plot in Figure 2 shows the distribution of the speedups achieved by FastKLEE. The labels under the *x-axis* correspond to the 40 utilities used in our evaluation, and the values on the *y-axis* represent the number of speedups by percentage. We can observe that most of the points in the scatter plot fall from 5% to 6%, and the highest point is up to 9.1%. Further, in Figure 4, we present the box plot depicting the distribution of the percentage number of the speedups achieved by FastKLEE in detail over 40 test programs. We could confirm that for the majority of the packages, the number of speedups ranges within [4.8%, 6.2%]. We also calculate the average of the speedups, which goes to 5.6% over those test programs. Interestingly, from Figure 2, it seems there is no discernible trend for the destruction of the speedups. This is because the benefits gained under FastKLEE may differ based on different test programs. Specifically, if a test program extensively checks *safe* pointers during execution, FastKLEE would be able to significantly reduce its checking overhead and greatly speed up the execution.

To further understand the overhead reduction achieved by FastKLEE, we also evaluate the time used for the type inference in FastKLEE. Figure 5 describes the trend of the time spent on CCured analysis over 40 test programs. We can see that the time speed of the analysis is within the range of [4.0s, 5.0s], which can be neglected compared with the whole time (usually taking hundreds or thousands of seconds to explore the 10k execution paths).

7 DISCUSSION

Potential implications of FastKLEE. Although we only show the performance benefits of FastKLEE in this paper, FastKLEE can have other important implications for path exploration in SE. For example, users can extend FastKLEE to assist SE to explore only valuable execution paths due to the time limit or path explosion challenge. The valuable paths can have at least two important forms. First, a path exploration strategy in FastKLEE can be guided

by the results of type inferences, meaning the paths that involve more *unsafe* pointers are more valuable (i.e., more likely to be buggy). That is, an *unsafe*-pointer-guided path exploration can be applied to explore valuable execution paths. Second, instead of targeting exploring multiple buggy execution paths, one may further leverage type inference results to explore the most valuable buggy path, i.e., paths that are more likely to be exploitable. Both the above implications can help improve the reliability and security of software systems. We leave this direction as our future work.

Threats to validity. One threat lies in the implementation of FastKLEE. We only implement FastKLEE on a source code-based symbolic executor. Since it is feasible to introduce a type inference system into binary code [4], we consider extending the support for other SE engines (e.g., Angr [24]) into FastKLEE in the future. Another threat comes from the test programs. We only used selected utilities in GNU `Coreutils`, and these programs may not be representative enough for various software systems. However, those test programs have been widely used for evaluating SE engines [5, 12, 18, 26, 28], and we consider expanding the program sets for more extensive evaluation in the future.

Limitation. FastKLEE has a limitation in terms of implementation, i.e., we only implement FastKLEE on top of a source code-based SE engine KLEE. Other binary code-based SE executors such as Angr [24] are not supported yet for the time we submit the paper. We plan to add the above support in future work.

8 CONCLUSION AND FUTURE WORK

We present FastKLEE, a tool that aims to reduce the interpretation overheads of redundant bound checking of type-safe pointers for faster SE. In FastKLEE, a type inference system is first leveraged to classify pointer types before interpretation. Then, a customized memory operation is designed to perform bound checking only for *unsafe* pointers during interpretation. Evaluation results demonstrate that FastKLEE outperforms the state-of-the-art KLEE in terms of the time spent on exploring the same number of execution paths. For future work, we are actively pursuing to (1) extend FastKLEE to support more SE engines and (2) leverage the abilities in FastKLEE to facilitate more comprehensive path exploration.

ACKNOWLEDGMENT

The authors would like to appreciate anonymous reviewers for their insightful comments. This article is partially supported by the National Research Foundation (NRF) Singapore and the National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) award number NSOE-TSS2019-04.

REFERENCES

[1] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57, 2 (2014), 74–84.
[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018), 39 pages.
[3] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 63–74.
[4] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4, Article 65 (2016), 35 pages.
[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 209–224.
[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. 380–394.
[7] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to Accelerate Symbolic Execution via Code Transformation. In *ECOOP*.
[8] 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.
[9] KLEE Official Document. 2022. Instruction for building GNU Coreutils. http://klee.github.io/tutorials/testing-coreutils/
[10] KLEE Official Document. 2022. Instruction for selecting running options. http://klee.github.io/docs/coreutils-experiments/
[11] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz 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.
[12] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2526–2540.
[13] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2022. DataGuard Repo.
[14] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2022. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the 2020 ISOC Network and Distributed Systems Security Symposium (NDSS'2022)*.
[15] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 177–187.
[16] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing Summaries of String Loops in C for Better Testing and Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 874–888.
[17] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. 193–204.
[18] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. *SIGPLAN Not.* 48, 10 (2013), 19–32.
[19] Daniele Midi, Mathias Payer, and Elisa Bertino. 2017. Memory Safety for Embedded Devices with NesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 127–139.
[20] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 128–139.
[21] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. 68–78.
[22] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.
[23] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS*.
[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157.
[25] STP. 2022. Simple Theorem Prover, an efficient SMT solver for bitvectors. (2022). https://github.com/stp/stp
[26] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A Bounded Symbolic-Size Model for Symbolic Execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1190–1201.
[27] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360.
[28] David Trabish and Noam Rinetzky. 2020. Relocatable Addressing Model for Symbolic Execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 51–62.
[29] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. Overify: Optimizing Programs for Fast Verification. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. 1–18.
[30] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 745–761.
[31] Z3. 2022. A theorem prover from Microsoft Research. (2022). https://github.com/z3prover/z3