Utilizing debug information to improve error-finding methods in the context of hybrid fuzzing

Larionov-Trichkine Theodor Arsenij
Information Security
National Research University "Higher School of Economics"
Moscow, Russia
tlarionovtrishkin@edu.hse.ru

***Abstract*—The use of fuzzing as a technique for identifying software vulnerabilities has become increasingly popular in recent years. While traditional fuzzing methods can be effective, they are inherently limited in their ability to find certain types of errors.**

Hybrid fuzzing tries to overcome these limitations by leveraging the power of symbolic execution to generate inputs that can explore different paths in a program. However, the effectiveness of hybrid fuzzing can be hindered by imprecise boundaries calculation for symbolic memory accesses. This research project aims to address this issue by utilizing debug information to improve error-finding methods in the context of hybrid fuzzing.

The core idea of the project is to enhance the calculation of boundaries for symbolic memory accesses by leveraging debug information. With more accurate boundaries, previously undiscovered bugs can be identified, and overall performance can be improved. The research will explore the feasibility of using debug information for boundaries calculation in the context of hybrid fuzzing and evaluate its effectiveness.

The project's methodology will involve implementing and testing the proposed approach on several open-source software projects. The results of this research will provide insights into the potential of using debug information to enhance hybrid fuzzing and could pave the way for more effective techniques in identifying software vulnerabilities.

***Index Terms*—Hybrid fuzzing, symbolic execution, symbolic pointers, symbolic addresses, symbolic memory model, debug information, error-finding, software security testing.**

I. Introduction

With the rapid development of information technologies, the number and complexity of software systems have increased drastically. This has led to an increase in the number of software vulnerabilities as well as an increasing need for secure software development practices.

One area of concern in software security is memory safety, particularly in the context of programming languages. Memory-unsafe languages, such as C and C++, are known to impose significant vulnerabilities due to their low-level nature and lack of automatic memory management [1], [2]. These vulnerabilities can range from buffer overflows to use-after-frees and beyond, and often lead to full system compromise.

To address these issues, a variety of techniques have been developed, including static and dynamic analysis. While static analysis is a powerful tool for identifying vulnerabilities, it has its limitations. To complement static analysis and improve the overall effectiveness of vulnerability detection, dynamic analysis techniques emerged.

Dynamic analysis often referred to as fuzzing, involves an automated generation of input data to test the behavior of a program at runtime.

The field of fuzzing has been actively developing for years, and nowadays, there is a variety of fuzzing techniques and tools available:

- AFL++ [3] is a popular fuzzing tool that uses a genetic algorithm and coverage feedback to generate new inputs.
- Superion [4] and Nautilus [5] are grammar-based fuzzers that use hand-crafted grammar to generate inputs with a specific structure.
- Sydr [6] and Fuzzolic [7] are examples of hybrid fuzzers that combine the power of symbolic execution with the speed of fuzzing.

All approaches are feasible, however, some of them require additional resources and time. That's where hybrid fuzzing shines. By using symbolic execution, it can generate structurally correct inputs that can explore different paths in a program without requiring any additional effort from the user.

Consider the following example:

```
1  void vuln(int key) {
2      if (key * 0x142a2d == 0xdeadbeef) {
3          error();
4      }
5  }
```

Listing 1. Example solvable by hybrid fuzzing

With traditional fuzzing, it is almost impossible to randomly generate an input that will trigger the error in the example 1. However, hybrid fuzzing methods allow us to generate a crash input by building a symbolic expression that represents the original input and then solving the constraints. This approach helps to overcome the limitations of traditional fuzzing and can be used to find a variety of bugs.

A hybrid fuzzing tool that I'm going to work with in this research is Sydr [6]. Sydr is a dynamic symbolic execution engine based on Triton [8] that is used to solve the programs' constraints and generate inputs that can explore new program states. Besides the ability to generate new inputs, Sydr also uses a novel technique called "Security Predicates" [9] to purposefully trigger error conditions in a program under test.

Sydr has a lot of features, and one of the most interesting ones is the ability to handle symbolic memory accesses. This feature allows Sydr to model symbolic accesses to memory and, as such, greatly improve the effectiveness of hybrid fuzzing. Consider the following example:

```
1  uint16_t crc_ibm_table[256] = {
2      0x0000, 0xc0c1, 0xc181, 0x0140, ...
3  };
4
5  uint16_t
6  crc_ibm_byte(uint16_t crc, const uint8_t c)
7  {
8      uint8_t sym_idx = (crc ^ c) & 0xFF;
9      return crc_ibm_table[sym_idx] ^
10         (crc >> 8);
11 }
12
13 if (crc_ibm_byte(0, buf[0]) == 0x1337) {
14     error();
15 }
```

Listing 2. Access to memory via a symbolic pointer

In the example 2, the program accesses memory via a symbolic pointer while calculating the checksum for a user-provided data **buf**. This means that the value of a symbolic pointer/index **sym_idx** depends on the input data. As a result, the program can access any element inside the **crc_ibm_table** array and this relation should be taken into account when building a symbolic expression for the program's constraints. Without correctly describing the memory access, the symbolic execution engine will not be able to generate an input that will lead to the error condition on line 14.

The current implementation of the symbolic memory model in Sydr is described in the paper: "Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution" [10]. It proposes a wide range of novel techniques to handle symbolic memory accesses. However, it still has room for improvement. For example, the current implementation of symbolic pointers uses a simple heuristic to determine the boundaries for symbolic memory access. However, this heuristic is not always accurate and might lead to incorrect results. Which in turn can hinder the effectiveness of hybrid fuzzing.

This research project aims to enhance the computation of symbolic memory access boundaries through the utilization of debug information. Debug information enables the identification of precise boundaries for various memory access types, such as local and global variables or heap objects. By obtaining more accurate boundaries, the Sydr can be improved in several ways:

- The effectiveness of Security Predicates can be enhanced as a result of the ability to detect additional types of bugs, such as local out-of-bounds bugs.
- The overall speed and accuracy of the Sydr can be improved since smaller memory regions exert less pressure on an SMT-solver.

II. Literature review

As previously discussed, the ability to handle indirect memory dependencies is a crucial feature in any hybrid fuzzing tool's effectiveness. Enhancements in this area can result in significant improvements in overall performance. In the following section, I will provide a concise overview of the leading methods for modeling symbolic memory, with a particular emphasis on one of the most crucial aspects of symbolic memory modeling: calculating precise boundaries for symbolic memory accesses.

A. Problem statement

The problem of precise boundaries calculation has been studied in the context of symbolic execution for a significant period of time. One of the key challenges in symbolic address reasoning is to determine the possible range of values that can be held by a symbolic address. To ensure accurate symbolic execution and avoid the loss of execution paths, it is crucial to identify the bounds that are closest to the actual ones. Failing to estimate the address range correctly can lead to imprecise symbolic execution and potential loss of execution paths. Conversely, an excessive address range can result in a larger symbolic expression for memory access, leading to higher memory consumption, increased SMT solver workload, and a decline in analysis performance.

B. Overview of the memory models

In a paper titled "A Survey of Symbolic Execution Techniques" [11], the authors provide a comprehensive overview of state-of-the-art symbolic execution techniques. The paper discusses the various approaches to symbolic memory modeling and provides a detailed comparison of the most popular methods. They examine several different approaches to symbolic memory modeling, including the following:

- Fully symbolic memory with state forking
- Fully symbolic memory modeled with ITE (if-then-else) formulas
- Address concretization
- Partial memory modeling
- Lazy initialization

Each of these methods has its own advantages and disadvantages. For example, address concretization shows the best performance in terms of execution speed, but it is not suitable for programs that use a large number of symbolic pointers, as it can lead to a significant loss of execution paths. On the other hand, fully symbolic memory modeling is the most accurate method, but it is also the slowest one.

C. Mayhem's approach

Mayhem [12] uses a different approach to model symbolic memory. It introduces an index-based memory model where memory is represented as a map $\mu : I \rightarrow E$ from 32-bit indices (i) to expressions (e). However, handling arbitrary symbolic indices is not always possible, as in general case a symbolic index can reference any memory location. That's why Mayhem uses a hybrid approach that models memory *partially*, where write operations are always concretized, but read operations are symbolic. This approach allows Mayhem

to handle a large number of symbolic pointers without losing too much performance.

To effectively utilize this approach Mayhem uses *memory objects*, which in most cases are significantly smaller than the entire memory μ. However, in order to implement this approach successfully, it is imperative to establish all potential values of a symbolic index i. In essence, this requires determining the precise boundaries of each symbolic memory access. To achieve this, Mayhem uses a slightly inaccurate method involving the use of an SMT-solver to approximate the boundaries of symbolic memory accesses.

This method shows manageable performance, however, it has several drawbacks:

- Solver queries are expensive and for 32-bit symbolic pointer access, you might need to perform ≈ 54 queries on average to determine both boundaries.
- The accuracy is not always sufficient.

D. Sydr's approach

The approach utilized by Sydr [6] involves a distinct technique that employs an AST-based heuristic to infer the lower bound. This approach entails examining the AST of the address expression and identifying the concrete segment of the address to deduce a lower bound. This method is described in the paper: "Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution" [10].

Consider an example with symbolic memory access: `mov ax, BYTE PTR [rdi + rsi * 0x4]`. Because the expression `rdi + rsi * 0x4` is symbolic, one of the operands (or all of them) must also be symbolic, in this case, it is `rsi`, the index for the array access. The heuristic algorithm will then analyze the AST of this expression and find the concrete part of the address, which is `rdi`. This part is then used as a lower bound.

This method is quite effective and allows to determine the lower bound of symbolic memory access with a high degree of accuracy while being relatively fast. However, it has some drawbacks:

- As the algorithm is based on the heuristic, it is not always possible to determine the lower bound.
- The accuracy of the algorithm is not always sufficient.
- And, probably the most significant drawback, is that the algorithm is not able to determine the upper bound, which substantially limits the opportunities for additional optimizations and improvements.

III. Methods

To improve the computation of symbolic memory access boundaries, I propose to utilize debug information. Debug information is a set of metadata that is used to provide information about the program's source code. Often, this information can be used to find exact sizes and boundaries for various memory regions. For example, debug information can be used to determine the size of global and local variables or the size of a heap object.

A general outline of the algorithm to compute the boundaries can be described as follows:

Data: mem_access - memory access to be analyzed
Data: left_boundary - leftmost boundary of the memory access
Data: right_boundary - rightmost boundary of the memory access
Result: left_boundary, right_boundary
$left\_boundary = 0$;
$right\_boundary = 0$;
if *mem_access is symbolic* **then**
  if *points to heap memory* **then**
    $info \leftarrow get\_chunk(mem\_access)$
  else if *points to global variable* **then**
    $info \leftarrow get\_global(mem\_access)$
  else if *points to stack (local) variable* **then**
    $info \leftarrow get\_local(mem\_access)$
  else
    Use default method to approximate boundaries
    $info.start \leftarrow find\_base(mem\_access)$
    $info.start \leftarrow info.start + offset$
  $left\_boundary \leftarrow info.start$
  $right\_boundary \leftarrow info.end$
end

Algorithm 1: Outline of the algorithm to compute the boundaries for symbolic memory accesses

By utilizing this algorithm, we can significantly enhance the accuracy of boundaries calculation. This is achieved by avoiding the need for approximations in many cases, and instead relying on the precise values obtained from debug information.

A. Metrics

To showcase the efficacy of the proposed approach, a series of preliminary experiments are underway. In assessing the ongoing enhancements, we are utilizing the following metrics:

$$\begin{aligned}\mathrm{Err}_i = |[\mathrm{Begin}_{\mathrm{default}}, \mathrm{End}_{\mathrm{default}}]| - \\ |[\mathrm{Begin}_{\mathrm{dbg}}, \mathrm{End}_{\mathrm{dbg}}]| \bigcap \\ [\mathrm{Begin}_{\mathrm{default}}, \mathrm{End}_{\mathrm{default}}]|\end{aligned} \tag{1}$$

$$\mathrm{Err} = \frac{\sum_{i=1}^{n} \mathrm{Err}_i}{n} \tag{2}$$

Metric 2 is a measure of how well the boundaries computed by the proposed approach match the boundaries computed by the default approach. The error is calculated as a ratio between the size of the intersection of the two sets of boundaries and the size of the default boundaries set. A higher error value means that the proposed approach differs significantly from the default approach. In other words, the proposed approach is able to compute more precise boundaries for a memory access, as the debug information gives the ground truth.

$$\mathrm{Error}_{\mathrm{left}} = \frac{\sum_{i=1}^{n} |\mathrm{LeftDbg}_i - \mathrm{LeftDefault}_i|}{n} \tag{3}$$

$$\text{Error}_{\text{right}} = \frac{\sum_{i=1}^{n} |\text{RightDbg}_i - \text{RightDefault}_i|}{n} \quad (4)$$

Left 3 and right 4 errors are measures of how well the proposed approach can compute the left and right boundaries of a memory access on average. The error is calculated as a ratio between the sum of the absolute differences between the left and right boundaries computed by the proposed approach and the default approach. When error values are high, it indicates a considerable deviation from the default approach. This suggests that the proposed method offers more precise results.

With this set of metrics, the effectiveness of the new approach can be evaluated on a set of benchmarks. Besides the metrics, a few different aspects also can be taken into account:

- Length of the formulas for the SMT solver to solve
- Overall performance
- Path accuracy
- Bug detection rate
- FuzzBench [13] benchmarks

B. Problems and limitations

While implementing the proposed approach, several problems and limitations were discovered. The most significant of them are the following:

- Location descriptions in debug information can be quite complex, which makes it difficult to extract the necessary information from them. This is especially a concern for highly optimized builds.
- In multimodule programs, finding the required information can be problematic due to the lazy loading of debug information.

Also, worth noting that this approach can't work in a black-box mode, since it requires debug information to be available. This means that the proposed approach can be used only for the programs that were compiled with debug information. However, this is not a significant limitation, since we assume that the programs that are analyzed by Sydr are compiled with debug information.

C. Notable advantages

The proposed approach in almost all cases can determine **the exact** boundaries for memory access without requiring any intensive computations. This is in contrast to the heuristic-based approach currently employed by Sydr, which is speedy but lacks accuracy and can't detect the upper boundary for memory access.

It also wins compared to the approach used by Mayhem [12], which utilizes an SMT-solver to find lower and upper boundaries for a memory access, as it is significantly faster and more accurate.

IV. Anticipated results

The results anticipated are based on already implemented techniques and the outcomes of preliminary experiments, as well as an analysis of the existing literature. The following improvements are expected:

- Enable more precise computation of symbolic memory access boundaries.
- The effectiveness of Security Predicates will be enhanced as a result of the ability to detect additional types of bugs, such as local out-of-bounds bugs.
- The overall speed and accuracy of the Sydr can be improved since smaller memory regions exert less pressure on an SMT-solver.

It should be noted, that some of the anticipated results are already achieved, but were not thoroughly evaluated or implemented. For example, a fragment of the proposed technique is already implemented as a part of the Sydr's Security Predicates and allows to detect local out-of-bounds bugs.

Eventually, after the final implementation of the proposed approach will be completed, the effectiveness of the Sydr will be evaluated on a set of real-world programs. The results will be presented in the final report.

V. Conclusion

The software security field is constantly evolving, being in a state of race between security researchers and hackers. Therefore, it's important to always stay up-to-date and detect new vulnerabilities before they are exploited.

In this paper, we presented an approach to greatly improve the capabilities of the hybrid fuzzing tool Sydr by utilizing debug information. We not only proposed a new technique to compute symbolic memory access boundaries but also implemented it and developed a set of benchmarks to evaluate its effectiveness.

It should be noted that the proposed approach has several notable advantages over the existing techniques:

- It greatly improves the accuracy of symbolic memory access boundaries calculation compared to heuristic and constant offset approaches.
- The performance of the method is remarkably better compared to the SMT-based approaches.

These enhancements promise a significant boost in the efficacy of error-detection methodologies, ultimately resulting in a reduction of exploitable vulnerabilities in the wild.

Further work on this topic can be done in several directions:

- Use debug information to find types of integers (signed/unsigned). With this information, it is possible to improve some aspects of the "Security Predicates" feature.
- Add support for recursive structure parsing. This will allow us to detect more bugs, such as out-of-bounds accesses inside structures. Besides, it will refine boundaries for symbolic pointers.
- Add better support for C++. Parse classes and their fields.

- Improve support for optimized builds by expanding the range of locations that can be processed. In addition to single-address or base-pointer offset location descriptions, it is possible to use more complex expressions, such as stack-pointer offsets.
- Use the known structure of an object to further optimize formulas for the SMT-solver.

REFERENCES

[1] "Chromium project memory safety report," https://www.chromium.org/Home/chromium-security/memory-safety/, accessed: 2023-02-12.

[2] "Android project memory safety report," https://source.android.com/docs/security/test/memory-safety, accessed: 2023-02-13.

[3] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[4] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," 2018. [Online]. Available: https://arxiv.org/abs/1812.01197

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[6] A. V. Vishnyakov, A. Fedotov, D. O. Kuts, A. A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky, and S. F. Kurmangaleev, "Sydr: Cutting edge dynamic symbolic execution," *CoRR*, vol. abs/2011.09269, 2020. [Online]. Available: https://arxiv.org/abs/2011.09269

[7] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: Mixing fuzzing and concolic execution," *Comput. Secur.*, vol. 108, no. C, sep 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102368

[8] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications*, ser. SSTIC, 2015, pp. 31–54. [Online]. Available: https://triton.quarkslab.com/files/sstic2015_slide_en_saudel_salwan.pdf

[9] A. V. Vishnyakov, V. Logunova, E. Kobrin, D. O. Kuts, D. Parygina, and A. Fedotov, "Symbolic security predicates: Hunt program weaknesses," *CoRR*, vol. abs/2111.05770, 2021. [Online]. Available: https://arxiv.org/abs/2111.05770

[10] D. O. Kuts, "Towards symbolic pointers reasoning in dynamic symbolic execution," *CoRR*, vol. abs/2109.03698, 2021. [Online]. Available: https://arxiv.org/abs/2109.03698

[11] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 380–394. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SP.2012.31

[13] J. Metzman, L. Szekeres, L. Maurice Romain Simon, R. Trevelin Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

Word count: 2563