Hotspot Symbolic Execution of Floating-Point Programs

Minghui Quan
College of Computer, National University of Defense Technology, Changsha, China
minghui_quan@163.com

ABSTRACT

This paper presents hotspot symbolic execution (HSE) to scale the symbolic execution of floating-point programs. The essential idea of HSE is to (1) explore the paths of some functions (called hotspot functions) in priority, and (2) divide the paths of a hotspot function into different equivalence classes, and explore as fewer path as possible inside the function while ensuring the coverage of all the classes. We have implemented HSE on KLEE and carried out extensive experiments on all 5528 functions in GNU Scientific Library (GSL). The experimental results demonstrate the effectiveness and efficiency of HSE. Compared with the baseline, HSE detects >12 times of exceptions in 30 minutes.

CCS Concepts

•Software and its engineering → Software verification and validation;

Keywords

Symbolic Execution; Floating point; Hotspot

1. INTRODUCTION

Floating-point programs are easy to have bugs [18]. Though symbolic execution [10] is often used in automatic testing [17] and bug finding [7, 3], existing symbolic executors [12, 3] barely support analyzing floating-point programs. The reason is the shortage of the backend solvers in floating-point solving. There exist a few SMT floating-point solvers [5, 4], but the scalability, usability and stability of using them in symbolic execution are still a problem.

This paper uses a lightweight method (denoted by $\mathcal{M}_L$) [15] to support the symbolic execution of floating-point programs. $\mathcal{M}_L$ uses an integer implemented floating-point library to replace floating-point operations. For example, when encountering a floating-point *add* instruction FAdd, symbolic executor executes an integer implemented *function* of FAdd (denoted by $\mathsf{Func}_{\mathsf{add}}$), instead of executing FAdd directly. Inside $\mathsf{Func}_{\mathsf{add}}$, no floating-point variable, expression or statement exists. In this way, floating-point expressions are converted into integer expressions. Without much effort, a symbolic executor (*e.g.*, SPF and KLEE) that *does not* support floating-point solving can be improved to analyze floating-point programs. Though $\mathcal{M}_L$ has the advantage in implementation, it naturally enlarges the space of paths, because each floating-point operation is converted to a function call. The effectiveness of the method may be embarrassed.

To evaluate $\mathcal{M}_L$, we have implemented it based on KLEE [3]. Extensive experiments are carried out on all 5528 functions in GSL [8]. Based on the evaluation, we propose *hotspot symbolic execution* (HSE) to tackle the path explosion problem, aiming to improve coverage further and detect more exceptions. The main idea of HSE is to: 1) explore the path space of a hotspot function in priority; 2) inside each hotspot function, the paths of the function are divided into equivalence classes, and are explored as fewer as possible while ensuring the coverage of each class. We have implemented HSE, and the results of extensive experiments demonstrate the effectiveness and efficiency of HSE.

The main contributions of this paper are as follows: 1) the algorithm of HSE, which scales symbolic executor by the balance between floating-point programs and integer implemented floating-point library; 2) the implementation and the extensive experiments on all 5528 functions in GSL, and the experimental results show that HSE detects >12 times of floating-point exceptions in 30 minutes, and achieves >2 times of the average function coverage in one minute on the GSL functions whose coverage is no more than 20%.

2. EVALUATING LIGHTWEIGHT METHOD

We have implemented $\mathcal{M}_L$ based on the improved version of KLEE (denoted by `KLEE-B`) in our group. `KLEE-B` implements under-constraint symbolic execution [6, 14] and lazy initialization [9]. The integer implemented floating-point library is SoftFloat [16]. This implementation is denoted as `KLEE-F`. The latest stable version of GSL [8] is used. Each GSL function is analyzed in 1, 10, 30 minutes.

Table 1: The result of evaluating `KLEE-F`

		Average Coverage Rate				#Exp
		≤ 100%	≤ 80%	≤ 50%	≤ 20%	
1m	#Funs	5528	2155	1367	442	
	KLEE-B	74%	42%	27%	13%	540
	KLEE-F	71%	43%	33%	20%	637
10m	#Funs	5528	2050	1317	422	
	KLEE-B	76%	42%	28%	13%	548
	KLEE-F	77%	52%	44%	29%	3958
30m	#Funs	5528	2001	1273	419	
	KLEE-B	76%	42%	27%	13%	588
	KLEE-F	78%	55%	46%	33%	5671

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983966

Table 1 shows the results of evaluation. The big column "Average Coverage Rate" shows the average coverage rates of different function sets. The column $\leq 100\%$ displays the average coverage rate of the functions whose coverage is less than or equal to 100% using KLEE-B, *i.e.*, all the functions. The numbers of the functions having different coverages are shown in #Funs rows. The column #Exp shows the number of detected exceptions. Four types of exceptions [1] are detected: Overflow, Underflow, Divide-by-Zero and Invalid.

Table 1 shows that KLEE-F and KLEE-B have an almost same average function coverage of all functions. The reason is GSL has many functions having no or fewer floating-point operations, and KLEE-B already performs well on these functions. When using 30 minutes, KLEE-F increases 20% average coverage for the functions whose coverage is no more than 20% with KLEE-B, and detects 9.6 times (5671/588) of exceptions, demonstrating the effectiveness of KLEE-F.

However, when using 1 minute, KLEE-F achieves minimum profit. The improvement in average coverage is less than 7%, and the number of exceptions only increases 77. The reason is KLEE-F often stucks into the functions in SoftFloat, which results in a small improvement or even decrease (*e.g.*, the average coverage of 5528 functions in 1 minute) when the analysis time is short. Hence, we need to guide KLEE-F to smartly explore the paths of the functions in SoftFloat.

3. HOTSPOT SYMBOLIC EXECUTION

Though symbolic executor needs to jump out from the functions in SoftFloat to avoid stucking, too fewer paths explored in the library may also produce a poor result. Hence, a balance is needed. The main intuition of HSE is that the behaviour of a function can be classified into different equivalence classes. In practice, one path in each class is enough.

The procedure of HSE is mainly the same with that of state-based symbolic execution [3]. If there is a hotspot function $\mathsf{F_H}$ executed, the states generated by $\mathsf{F_H}$ will be explored in priority. The exploration inside $\mathsf{F_H}$ will be guided by the information of equivalence classes of $\mathsf{F_H}$ to produce states as fewer as possible while ensuring the coverage of all the classes. After covering all the classes, the remaining states of $\mathsf{F_H}$ will be explored in a lower priority, making symbolic executor leave $\mathsf{F_H}$ quickly.

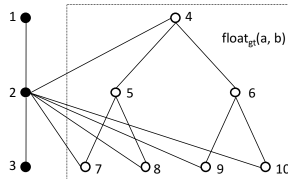

Figure 1: Example of hotspot execution

Take the greater than $(a > b)$ function $\mathsf{float_{gt}}(a, b)$ for an example. Figure 1 shows the hotspot execution of $\mathsf{float_{gt}}(a, b)$. A circle represents a state in symbolic execution. A method call to $\mathsf{float_{gt}}(a, b)$ is executed at state 2. $\mathsf{float_{gt}}(a, b)$ has four paths: 4-5-7 (p_1), 4-5-8 (p_2), 4-6-9 (p_3) and 4-6-10 (p_4), divided into two equivalence classes (denoted by c_1 and c_2) representing $true$ and $false$ cases, respectively. p_1 and p_2 belong to c_1, and the rest two belong to c_2. At the beginning, state 4 will be explored in priority. Inside $\mathsf{float_{gt}}(a, b)$, the states will be *randomly* selected. If the first finished path is p_1, a state will be selected from states 8 and 6. Suppose state 6 is selected and then p_3 is finished, which makes c_2 be covered. Then, c_1 and c_2 are all covered, so the priorities of the remained states, *i.e.*, 8 and 10, are lowed. Next, symbolic executor will explore the states outside of $\mathsf{float_{gt}}(a, b)$.

HSE is implemented based on KLEE-F. The hotspot functions are those in SoftFloat corresponding to the *comparison*, *arithmetic* and *converting* LLVM floating-point instructions. This implementation is denoted as KLEE-HS, whose evaluation is shown in the following table.

Table 2: The result of evaluating KLEE-HS

		Average Coverage Rate				#Exp
		$\leq 100\%$	$\leq 80\%$	$\leq 50\%$	$\leq 20\%$	
1m	#Funs	5528	2540	1450	411	
	KLEE-F	71%	43%	28%	12%	637
	KLEE-HS	71%	49%	36%	25%	1768
10m	#Funs	5528	2172	1100	202	
	KLEE-F	77%	48%	30%	12%	3958
	KLEE-HS	76%	53%	39%	20%	5660
30m	#Funs	5528	2070	1012	184	
	KLEE-F	78%	49%	31%	12%	5671
	KLEE-HS	77%	54%	40%	20%	7087

Table 2 indicates KLEE-HS outperforms KLEE-F in most cases. In one minute, KLEE-HS increases the rate from 12% to 25% (>2 times) on the functions whose coverage is no more than 20% using KLEE-F, and detects 2.7 times (1768/637) of exceptions. It demonstrates the effectiveness of HSE. Even with a longer time, KLEE-HS can detect more exceptions (1702/1416 in 10/30 minutes).

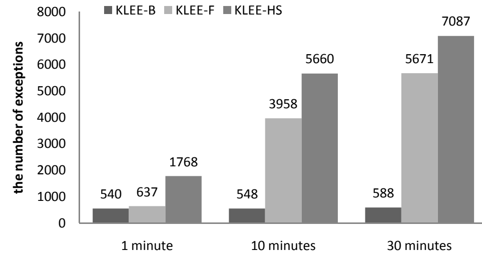

Figure 2: Detected exceptions

Figure 2 shows KLEE-HS detects more exceptions. When using 30 minutes, KLEE-HS detects >12 times of exceptions than KLEE-B. For all the three, the number of detected exceptions (denoted by #) increases with the increase of time, but the increase of #KLEE-B is slow, which means increasing time to detect more exceptions is not effective without floating-point support. #KLEE-HS/#KLEE-B increases when increasing the time, which demonstrates the effectiveness and efficiency of KLEE-HS. #KLEE-HS/#KLEE-F decreases with the increasing of time, indicating KLEE-HS is more effective than KLEE-F to detects exceptions when the analysis time is short.

4. RELATED WORK

Closely related work can be divided into two categories: 1) use the SMT solver with floating-point support for symbolic execution [13]; however, it is a problem for SMT solvers to support mixed types of constraints, and the integration of solvers also needs engineering effort. 2) convert floating-point expressions into rational number expressions, and use the SMT solver with rational number support for solving [2, 11, 1]. Compared with these work, our approach is more lightweight, easier to implement and more stable. The experimental comparison with the existing work, such as [1], is interesting and left to be the future work.

ACKNOWLEDGMENTS

This work was supported by National 973 (2014CB340703) and NSFC (61472440, 61632015, 61272140) of China.

5. REFERENCES

[1] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560. ACM, 2013.

[2] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2):97–121, 2006.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[4] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*. Springer, 2013.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340. Springer, 2008.

[6] D. R. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA*, pages 1–4. ACM, 2007.

[7] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[8] GSL 2.1. GNU Scientific Library (GSL). http://www.gnu.org/software/gsl/.

[9] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568. Springer, 2003.

[10] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. Flopsy - search-based floating point constraint solving for symbolic execution. In *ICTSS*, pages 142–157, 2010.

[12] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, pages 15–26. ACM, 2008.

[13] J. Ramachandran, C. S. Pasareanu, and T. Wahl. Symbolic execution for checking the accuracy of floating-point programs. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.

[14] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *SEC*, pages 49–64. USENIX Association, 2015.

[15] A. Romano. Practical floating-point tests with integer code. In *VMCAI*, pages 337–356. Springer, 2014.

[16] SoftFloat 2b. Berkeley SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html.

[17] N. Tillmann and J. De Halleux. Pex–white box test generation for.NET. *Tests and Proofs*, pages 134–153, 2008.

[18] Wikipedia. Ariane 5 flight 501. http://en.wikipedia.org/wiki/Ariane_5_Flight_501.