CLEVER-C: Extending CLEVER to Include Coverage of C Programs

Alexander Tough, Federico Mora and Marsha Chechik
University of Toronto
Department of Computer Science
alex.tough@mail.utoronto.ca, fmora@cs.toronto.edu, chechik@cs.toronto.edu

I. INTRODUCTION

Software systems are often composed of multiple related but independently developed components. A change in one component could have repercussions in other parts of the system. Hence, dealing with component upgrades can become a time-consuming task for software testers.

One way to manage this problem, as shown by Mora et al.[1], is to use logic reasoning to determine which parts of a codebase (called a *client* program) are affected by a library update. This is relevant in practice, since for example the open source projects Delorean, OpenSSL, Linux and GMP, 71% of the client programs of these projects remain unaffected by the changes to their library functions. Mora et al. introduce the notion of *client-specific equivalence* (CSE) to determine whether library changes affect individual clients. They also propose a general algorithmic framework called CLEVER: Client Specific Equivalence Checker.

Symbolic execution is a program analysis technique that simultaneously explores the multiple paths that a program could take under different inputs by allowing it to take on symbolic – rather than concrete – values[2]. CLEVER reduces the CSE checking problem to the validity of first-order formulas via symbolic execution.

Partial functional equivalence of two programs P1 and P2 can be defined as whether any two terminating executions of P1 and P2 return the same value given the same inputs [3]. The original implementation of CLEVER (CLEVER-PyExZ3) described in [1] performs well against state-of-the-art tools [4] [5] [6] [7] that check for partial functional equivalence on a set on non-trivial benchmarks. However, this implementation of CLEVER does present several issues, including:

- It only accepts Python programs as input, meaning that all C input programs must be translated to Python. This is because CLEVER-PyExZ3 is built on top of PyExZ3 [8], a symbolic execution engine for Python written in Python, and PySMT [9], a Python interface to SMT solvers. This is also the reason why the tool is limited to integer reasoning [1].
- There is no support for heap manipulations, floating point numbers, strings and objects composed of these primitives [1].

II. APPROACH

Of the limitations listed above, the most pressing one is the Python-only input, since most of the code in the OpenSSL, Linux and GMP – and countless other open-source projects – is written in C. To solve this issue, instead of using PyExZ3, as done in CLEVER-PyExZ3, we opted to use KLEE [10] as the symbolic execution engine for this version of CLEVER (CLEVER-C). KLEE can process C programs and implements shadow symbolic execution[11]. This flavour of symbolic execution is ”a technique for generating inputs that trigger the new behaviours introduced by a patch”[11], which is relevant to CSE. For that reason and because it is a state-of-the-art symbolic execution engine, KLEE was a natural choice.

III. ANALYSIS

We evaluated CLEVER-C and CLEVER-PyExZ3 on 14 benchmarks, each consisting of a pair of programs before and after some changes in the library. They were a subset of the same ones tested on the previous implementation of CLEVER. In total, 9 pairs were equivalent and 5 pairs were inequivalent.

We performed the experiments on CLEVER-C on an Intel Core i5 1.8 GHz CPU with 8 GB of memory running macOS High Sierra (10.13.6), running a Ubuntu 14.04 virtual image using Docker. CLEVER-PyExZ3 was run on an Intel Core i7 4.00 GHz CPU with 16 GB of memory running Windows 10 with Cygwin, as reported in [1]. For each benchmark in both versions, we set a timeout of 300 seconds.

Table I shows the results of the comparison between the two versions of CLEVER. In all of the cases considered, CLEVER-C fared worse than CLEVER-PyExZ3. This could be attributed to the fact that our implementation of CLEVER does not include key optimizations present in the previous version, including support for uninterpreted functions and incremental lazy exploration [1]. Those two optimizations reduce the number of paths to explore during symbolic execution. Not implementing these optimizations allows us to have a baseline version of CLEVER without dealing with the specifics of the optimizations, which can be cumbersome when migrating from one symbolic execution engine to another, as we have done in this implementation.

Another important factor that could have increased execution time of CLEVER-C is I/O. According to profiling, on average 21% of runtime was spent doing file I/O. This is due to a number of temporary files created and pipes used to connect the inputs and outputs of the the individual components of the CLEVER framework.

TABLE I: Run-time in seconds of CLEVER-C and CLEVER-PyExZ3 where "–" indicates that the tool either times out or reports inconclusive results.

	Benchmarks	CLEVER-C	CLEVER-PyExZ3
Equivalent	factorial	1.6	**0.295**
	fib	×	**0.268**
	get_sign2	1.11	**0.068**
	is_prime1	1.13	**0.056**
	is_prime3	×	**1.289**
	ltfive	1.17	**0.108**
	multiple	2.32	**0.077**
	pos2	–	–
	pos3	1.17	**0.085**
Non-Equivalent	factorial	69.10	**0.081**
	fib	1.41	**0.216**
	get_sign2	1.31	**0.047**
	loopunreach2	1.34	**0.062**
	loopunreach20	1.31	**0.075**

IV. Conclusion

In this paper, we presented CLEVER-C, which substitutes CLEVER-PyExZ3's choice of symbolic engine (PyExZ3) for KLEE. We compared our implementation against the previous implementation on a set of non-trivial benchmarks.

In future work, we intend to reduce the overhead introduced by file I/O at all stages of the framework. We also intend to extend CLEVER-C not only to handle integers, but other types of numeric data. We also intend to modify the KLEE engine so that incremental lazy evaluation [1] can be performed, and modify CLEVER-C to include support for uninterpreted library calls [1]. Following the long-term goals detailed in [1], we also intend to extend coverage to heap manipulations and add support for floating point numbers, strings, and objects composed of those primitives.

References

[1] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, 2018.

[2] 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, 2018.

[3] B. Godlin and O. Strichman, "Inference rules for proving the equivalence of recursive procedures," *Acta Informatica*, vol. 45, no. 6, pp. 403–439, 2008.

[4] ——, "Regression verification: Proving the equivalence of similar programs," *J. of Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 241–258, 2013.

[5] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *Proc. of SIGSOFT FSE '08*, 2008.

[6] A. Trostaneski, O. Grumberg, and D. Kroening, "Modular demand-driven analysis of semantic difference for program versions," in *Proc. of SAS '17*. Springer, 2017, pp. 405–427.

[7] D. Felsing, S. Grebing, V. Klebanov, P. Rummer, and M. Ulbrich, "Automating regression verification," in *Proc. of ASE '14*. ACM, 2014, pp. 349–360.

[8] T. Ball and J. Daniel, "Deconstructing dynamic symbolic execution," in *Proc. of the 2014 Marktober Summer School on Dependable Software Systems Engineering*. IOS Press, 2015.

[9] M. Gario and A. Micheli, "pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms," in *SMT Workshop*, 2015.

[10] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of OSDI'08*, 2008.

[11] C. Cadar and H. Palikareva, "Shadow symbolic execution for better testing of evolving software," in *Proc. of ICSE'08*, 2014.