Comparing Application Semantics with S^{2}E

Determining whether busybox and coreutils are actually the same

Mariana D’Angelo
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
mariana.dangelo@utoronto.ca

Dhaval Miyani
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
dhaval.miyani@utoronto.ca

I. Introduction

While the key idea behind symbolic execution arose almost forty years ago [3], it has only recently become practical due to advances in constraint satisfiability [4] and scalable approaches with mixed symbolic and concrete execution [5], [16]. A 2011 study has shown that symbolic execution tools are beginning to be used increasingly in industrial practice at corporations such as Microsoft, NASA, IBM, and Fujitsu [2]. This proliferation shows the importance of new symbolic execution tools and the impact they will have on the industry as they improve.

S^{2}E is a symbolic execution platform that can operate directly on application binaries for analyzing the properties and behaviour of software systems. It analyzes programs in-vivo (the whole environment) within a real software stack (user program, libraries, kernel, drivers, etc.) rather than using abstract models of these layers. It is commonly used for performance profiling, reverse engineering of proprietary software, and finding bugs in user-mode and kernel-mode binaries [1]. S^{2}E is a complex system due to the rich features it has (particularly the fact that it runs inside of a VMM). This complexity, however, comes with the price of being difficult to start using on existing software.

The goal of this experiment is to understand how a complex system such as S^{2}E can be applied to existing software and document the process. In particular, we are interested in seeing whether program inputs and code need to be sanitized in some way to become amenable to dynamic analysis via symbolic execution. Taking inspiration from KLEE [6], we will be comparing the semantics of two implementations of (supposedly) the exact same suite of programs: the GNU Coreutils utility suite and the Busybox embedded system suite.

Comparing the semantics of two implementations can have many real world applications. In the field of computer security, for example, the ability to compare the semantics of a software compiled from source with a pre-compiled binary could reveal malicious or accidental backdoors. In particular, the fact that S^{2}E runs in a VMM would allow for it to be used to test kernel level programs such as file systems (or compare read and write operations different file systems).

We have encountered several challenges in using the S^{2}E platform; one in particular forced us to change the initial goal of our experiment (evaluating two consistency models from S^{2}E) due to the fact that not all of the six consistency models described in [1] are actually implemented in the platform. Our biggest challenge with this project was trying to modify the programs of interest in such a way that they could be directly comparable. The different approaches we tried will be thoroughly discussed in the Experiments section of this paper along with our own explanations about why they failed or succeeded and what impact these choices had on the program analysis as a whole.

This paper is organized as follows: Section II presents some background regarding symbolic execution, Section III discusses our approach to the experiments, Section IV describes in detail how we attempted to implement our approach, Section V discusses the results of our successful experiment with echo, Section VI presents the status of our implementation, Section VII concludes with some lessons learned, and finally in Section VIII we discuss future work.

II. Background

There is much work in symbolic execution for testing [6], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [7]. A typical approach for testing is fuzzing [17], which involves generating random inputs for a program in an attempt to exercise all paths in a program (and hopefully hit any buggy code). This is known as concrete execution which involves running the program with deterministic values. Fuzz testing has several limitations, in particular coverage of paths in a program. For example, Listing 1 shows a if statement which is only taken when x is 10. There are 2^{32} possible values which x could take when an input is randomly generated and as such the probability of actually taking this path is 2^{-32}. This example demonstrates that the likelihood of some paths being taken when fuzzing is extremely low, which is the reason for the poor test coverage of concrete execution.

```
if (x == 10) {
    // path 1
} else {
    // path 2
}
```

Listing 1. Code example where fuzzing generally has poor coverage

Symbolic execution, on the other hand, runs the application with symbolic inputs [6] which are initially unconstrained by design. The program executes with these symbolic values and replaces concrete operations with ones that can manipulate symbolic values. The symbolic execution engine "follows" both paths whenever it encounters a branch on a symbolic value and adds the path condition to a set of constraints known as the "path constraint". When a bug is encountered in the code a test case (i.e., concrete inputs) can be generated from the path constraint. There are two main limitations of symbolic execution: (1) "the path explosion problem" [1], which is due to the exponential growth of paths in a program caused by conditional branches, and (2) "the environment problem" [6], which is due to interactions with the surrounding environment (e.g, operating system, network, etc.).

In order to deal with the environment problem, one can use a mixture of symbolic and concrete execution (also known as concolic execution [7]). Concolic execution gathers constraints using symbolic execution, but then generates concrete values using a constraint solver in order to address the environment problem. Once these concrete values are generated the program can interact with its environment in a normal fashion without the overheads and problems associated with symbolically executing the environment.

S^2E is a symbolic execution engine which can perform concrete execution, symbolic execution, and concolic execution depending on the consistency model chosen. There are six different consistency models in S^2E, described below. Figure 1 provides an overview of the models showing how they transition from highly strict to highly relaxed models with some details about what consistency requirements are relaxed for each model. A model is consistent if there exist a globally feasible path through the system for every path explored in the unit. A unit is the block one wishes to analyze and the environment is the rest of the system.

- *Strictly Consistent Concrete Execution (SC-CE):* No symbolic execution in unit or environment.
- *Strictly Consistent Unit-level Execution (SC-UE):* Unit is symbolically executed while the environment is executed concretely.
- *Strictly Consistent System-level Execution (SC-SE):* Unit and environment are executed symbolically - this is the only model that executes the environment symbolically.
- *Local Consistency (LC):* Similar to SC-UE, but it adheres to constraints that the environment/unit API contracts impose on return values.
- *Relaxed Consistency Overapproximate Consistency (RC-OC):* Similar to LC, but it ignores the constraints from the environment/unit API contracts.
- *Relaxed Consistency CFG Consistency (RC-CC):* Similar to SC-UE, but like static analysis it can explore any path in the unit's inter-procedural control flow graph (even infeasible ones).

There are several differences between S^2E and other similar symbolic execution engines. For instance, KLEE uses file system models [6] to avoid symbolically executing the actual filesystem. S^2E does not take this approach as writing models is a labour intensive and error-prone process. Cute [7] executes the environment concretely (i.e., without modelling) with a

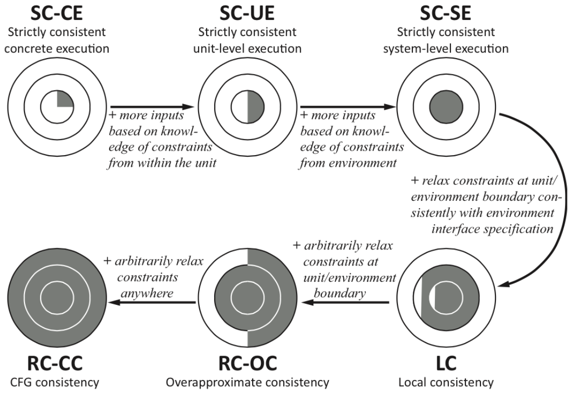

Fig. 1. S^2E Consistency Models [1]

consistency model similar to S^2E's SC-SE, but it is limited to code-based selection and one consistency model. S^2E does not use compositional symbolic execution [15], a performance optimization which saves results for parts of the program (e.g., a function) and reuses them when that part is called in a different context. With concolic execution everything runs concretely for full systems, however, when the execution crosses program boundaries it may result in lost paths [1]. Due to this, KLEE and CUTE cannot track path conditions in the environment and hence are unable to re-execute calls to enable overconstrained but feasible paths (e.g., malloc does not execute deterministically). DART [16], CUTE [7] and EXE [11] use mix-mode execution (concretely executing some parts and symbolically executing others) to increase efficiency, however, they do not use automatic symbolic-concrete bidirectional data conversion (i.e., automatic conversion between concrete and symbolic values at program boundaries such as the unit and the environment) unlike S^2E, which is key to S^2E's scalability and low programmer effort.

Static analysis is performed without executing the program in question and has known limitations such as a high false positive and negative rate (due to infeasible paths) [17] and a lack of runtime environment analysis (as only the application source code is parsed). To reduce the number of false positives tools such as Saturn [19] and bddbddb [20] use a path-sensitive analysis engine. Saturn aims to detect logic programming language bugs by summarizing functions. bddbddb finds buggy patterns in a database where programs are stored as relations. As can be inferred, these tools are language specific and require different implementations for different programming languages. Additionally, using these tools requires learning a new programming language. As a dynamic analysis framework, S^2E addresses the limitations of static analysis tools. For example, they directly operate on and analyze binaries whereas static analysis would require disassembly and decompilation. This could involve converting an x86 binary to the LLVM format and running it through an engine like KLEE. Disassembly and decompilation [18] are classically undecidable problems (e.g., disambiguating code from data) [1].

III. Approach

The high level approach we took to use S2E for equivalence testing was as follows:

1) Run each program with symbolic input (using `s2e_make_symbolic()` from a test application invoking the functions or directly on the binary with option `--sym-args`).
2) Compare the output of the Busybox and GNU Coreutils implementation of the same program using `s2e_assert`, which performs a regular assert but also does some cleaning up for S2E and generates sample input which violates the assertion.
3) Using the sample input generated, manually analyze the two implementations (possibly using GNU Debugger) to identify semantic differences between the two implementations.

IV. Implementation

In this section we will describe all of the ways in which we tried to symbolically execute the two binaries (from GNU Coreutils and Busybox) to perform equivalence testing.

A. Running the binary directly

Initially, we attempted to run the analysis directly on the binaries for the two implementations of echo - having built the two utility suites precisely as suggested by the developers. These results, however, were not easily comparable and did not provide information regarding the semantic differences between the two implementations. Listing 2 shows the (abridged) results of symbolically executing GNU Coreutils' implementation of `echo`. From this attempt, it was apparent that running S^{2}E directly on a binary is better suited for simply testing an application with all possible inputs and high coverage, rather than determining semantic equivalence.

```
PUT A LISTING HERE WITH THE OUTPUT OF EXECUTING
    S2E DIRECTLY ON THE ECHO BINARY
```

Listing 2. Output of symbolically executing GNU Coreutils `echo` with S^{2}E

B. Creating libraries for each implementation

[DHAVAL] Tried to make the echo implementations into shared libraries so they could be called directly from our test program (shared library vs. static library stuff)

C. Redirecting stdout to a buffer

[MARIANA] Once we had these applications we tried simply redirecting stdout into a buffer to read their outputs (failed because busybox does not use stdio).

After all of these attempts we decided to look into how KLEE ended up doing this comparison of coreutils and busybox. We discovered that KLEE was symbolically modifying stdout (klee option `-sym-stdout`). Since S2E is built upon KLEE we tried to add this flag into the klee arguments portion of the config.lua file for S2E (include listing with config file we used for our testing? or is it not relevant because testcasegeneration didn't even seem to work...), however this option did not exist, most likely due to the fact that s2e is meant to be in vivo and use a real software stack (i.e., not model any of those layers).

D. Modifying the implementations to not use stdout

[DHAVAL] Final approach: modifying coreutils echo and busybox echo to return a string instead of printing to stdout [SUCCESS!]

V. Experiments

[MARIANA]

- modified echo to return strings for both implementations
- got 4 assertions that failed between the two implementations
- manually inspected the programs with gdb given the inputs provided by s2e (e.g., 0x2c02) to step through the execution and see where execution differed
- We even discovered a bug in our own modifications to coreutils echo where we were adding a space instead of newline!
- each time we run s2e we get a different number of states even though they should hypothetically always go through the same number of states
- one example of a difference between the implementations of echo is that both echo's print a space between strings found in argv, however the condition preceding the addition of this space is different between the implementations (for coreutils) if there is at least one string to print (check argc ¿ 0), a space is always added after the word whereas busybox only prints a space after a word if there is another string to print (breaks if *argv++ not NULL).
- two input patterns that failed the asserts (1) ended with 0x00 (i.e., NULL character), and (2) ended with extended ascii values (i.e., expressed as a signed integer -0x45 which is a symbol)

VI. Status

[DHAVAL] Current status of implementation.

- After several failed attempts we decided to modify the utilities and got S2E working with both implementations of echo.

VII. Conclusion

[DHAVAL] Did your results meet expectations?

- Summarize intro (what did we do, why did we do it)
- Lessons learned: lack of documentation and an incomplete understanding of how S2E symbolically executes led to a lot of mistakes.

VIII. Future Work

[DHAVAL]

- Perform these tests on the rest of the utilities in busybox and gnu coreutils.
- Try testing other programs that don't use stdout and determine if they present any different challenges.
- Figure out how to change the VMM parameters for S2E (e.g., give it more memory). Discuss how even building from source had its problems as we could not transfer files from host to guest when we built S2E manually.

References

[1] CHIPOUNOV, V., GEORGESCU, V., ZAMFIR, C., AND CANDEA, G Selective symbolic execution. In Workshop on Hot Topics in Dependable Systems, 2009.

[2] CADAR, C., GODEFROID, P., KHURSHID, S., PASAREANU, C., SEN, K., TILLMANN, N., AND VISSER, W. Symbolic execution for software testing in practice preliminary assessment. In ICSE Impact11, May 2011.

[3] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. SELECT - a formal system for testing and debugging programs by symbolic execution. SIGPLAN Not., 10:234245, 1975.

[4] DE MOURA, L. AND BJRNER, N. Satisfiability modulo theories: introduction and applications. Commun. ACM, 54:6977, Sept. 2011.

[5] CADAR, C. AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In SPIN05, Aug 2005.

[6] CADAR, C., DUNBAR, AND ENGLER, D.R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Symp. on Operating Systems Design and Implementation, 2008.

[7] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A concolic unit testing engine for C. In In 5th joint meeting of the European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE 2005).

[8] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking path explosion in constraint-based test generation. In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008).

[9] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based sig- natures. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (IEEE S&P 2006).

[10] CADAR, C., AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In Proceedings of the 12th International SPIN Workshop on Model Checking of Soft- ware (SPIN 2005).

[11] CADAR, C., GANESH, V., PAWLOWSKI, P., DILL, D., AND ENGLER, D. EXE: Automatically generating inputs of death. In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006).

[12] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing software by blocking bad input. In Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP 2007).

[13] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: end-to-end containment of Internet worms. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005).

[14] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic test input generation for database applications. In International Symposium on Software Testing and Analysis (ISSTA 2007).

[15] GODEFROID, P. Compositional dynamic test generation. In Proceedings of the 34th Symposium on Principles of Programming Languages (POPL 2007).

[16] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005).

[17] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In Proceedings of Network and Distributed Systems Security (NDSS 2008).

[18] SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. Disassembly of executable code revisited. In Working Conf. on Reverse Engineering, 2002.

[19] DIILLIG, I., DILLIG, T., AND ALKEN, A. Sound, complete and scalable path- sensitive analysis. In Conf. on Programming Language Design and Implementation, 2008.

[20] LAM, M. S., WHALEY, J., LIVSHITS, V. B., MARTIN, M. C., AVOTS, D., CARBIN, M., AND UNKEL, C. Context-sensitive program analysis as database queries. In Symp. on Principles of Database Systems, 2005.