Finding the Right Consistency Model

A comparison of the different consistency models in S2E for testing

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

S2E is a symbolic execution platform that operates directly on application binaries for analyzing the properties and behavior 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].

The goal of this experiment is to evaluate the performance of the consistency models in the S2E system when finding different types of bugs. We are interested in investigating whether we can identify program features (e.g., program length, bug type, etc.) that pose challenges for certain consistency models or present benefits. Some software systems are only exposed to certain kinds of bugs (e.g., a batch processing script will not be exposed to concurrency bugs). This observation leads us to believe that a given consistency model may fit specific a application domain better than others.

As different consistency models treat the environment differently we expect those that symbolically execute the environment will perform slower and have higher memory consumption [2]. As stated previously certain consistency models permit inconsistency in the environment and we expect that bugs which manifest on changes in the environment will not be detected by these models. The SC-SE model suffers from path explosion [1] and as such we expect it to have the worst performance overall, though it should catch bugs given unlimited time and resources.

Although [1] recommends the use of the RC-CC model we expect that there will be certain cases in which other models outperform it. If this holds true, we will perform an analysis to determine what the causes are and if certain program features trigger the performance differences.

II. Introduction

stating purpose of the project, expected outcome or result of the project

- Introduction: stating purpose of the project, expected outcome or result of the project
- Purpose/Goal - Primary objective is to determine the accuracy of the Relaxed Consistency models. Secondary obj is to observe the performance
- Find the trade-off between the accuracy vs Performance For e.g. running for another 2 additional hours might increase the accuracy by 10% or 1%?!
- Expected outcome - ?The main advantage of RC is performance: by admitting these additional infeasible paths, one can avoid having to analyze large parts of the system that are not really targets of the analysis, thus al- lowing path exploration to reach the true target code sooner. How- ever, admitting locally infeasible paths (i.e., allowing the internal state of the unit to become inconsistent) makes most analyses prone to false positives, because some of the paths these analyses are ex- posed to cannot be produced by any concrete run.? [S2E paper]

III. Background

There is much work in symbolic execution for testing [CITATIONS FROM KLEE 11, 14−16, 20−22, 24, 26, 27, 36]. A typical approach for testing is fuzzing [CITE REF], 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 [MAYBE CITE?]. Fuzz testing has several limitations, in particular coverage of paths in a program. For example, [figure 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
}
```

Fig. 1. Code example where fuzzing generally has poor coverage

[ADD FIGURE LABEL, ETC.]

Symbolic execution, on the other hand, runs the application with symbolic inputs [2] 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” [CITE], which is due to the exponential growth of paths in a program caused by conditional branches, and (2) ”the environment problem” [2], 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 [3]). 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.

S2E 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 S2E, described below. 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).

Our experiment will focus on evaluating the two latter models: RC-OC and RC-CC. These models are of particular interest because we cannot accurately predict the accuracy or performance of these two models in contrast to the other four models [1].

There are several differences between S2E and other similar symbolic execution engines. For instance, KLEE uses file

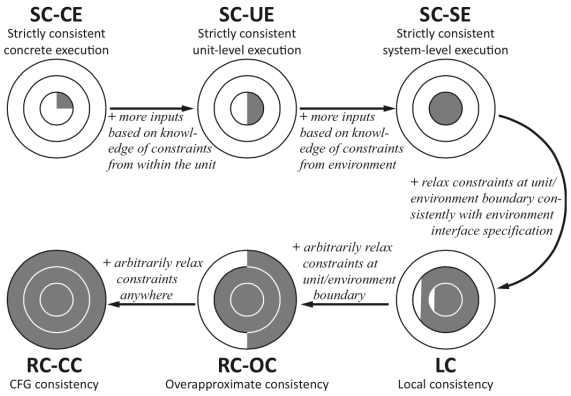

Fig. 2. S2E Consistency Models [1]

system models [2] to avoid symbolically executing the actual filesystem. S2E does not take this approach as writing models is a labour intensive and error-prone process. Cute [3] executes the environment concretely (i.e., without modelling) with a consistency model similar to S2E's SC-SE, but it is limited to code-based selection and one consistency model. S2E does not use compositional symbolic execution [11], 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 [12], CUTE [3] and EXE [7] 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 unlike S2E, which is key to S2E'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) [13] and a lack of runtime environment analysis (as only the application source code is parsed). To decrease the number of false positives rate tools such as Saturn [15] and bddbddb [16] 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 require to learn a new language. As a dynamic analysis framework, S2E 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 [14] are classically undecidable problems (e.g., disambiguating code from data) [1].

OUTLINE:

- Symbolic Execution: KLEE, SAGE, DART, Execution Synthesis what Symbolic Execution is?
- Static Analysis: ¡insert paper here¿

- Performance vs. Accuracy tradeoff: ¡insert paper here¿

IV. Approach

research methodology or approach taken in the project

A. Sanity Check

- Write a few short (¡ 100 LOC) dummy programs with different bugs (e.g., null pointer dereference, divide by zero, buffer overflow, etc.) to ensure that S2E can catch these bugs before evaluating larger programs - determine coverage of the consistency models before starting.

B. Dataset

- Select three programs: (1) Web server like apache - lots of network usage (syscalls), (2) Database like SQLlite - lots of file i/o (syscalls), (3) Graphics editing software like Inkscape - lots of computation (not many syscalls expected)

C. Ground Truth

- Look at bug reports for current (or older) versions of these programs to determine what bugs we expect S2E to find (?ground truth?-ish) and evaluate whether we should inject our own bugs.

D. Measurement

- Use a script or something to get the time of execution start to bug finding (somehow) - check if S2E does logging already (in that case we can just check what time that was added or whatever). Also need to write a script which cuts off S2E after certain time units.

V. Status

current status of implementation

- Currently in preparatory phase
- We have built S2E and tested it on a sample application
- Gathering dataset, looking at bug reports for each application
- Wrote dummy programs with bugs to ensure the consistency models catch these bugs (sanity check)

VI. Evaluation

description of experiments that will be performed

- Per consistency model: Using script described above we will run each of the three applications for different ?time units? (e.g., one hour cutoff, two hour, ?., five hour) and record results (hopefully S2E has a log?..)
- Analyze bugs caught, comparing them to the ?ground truth? determined previously - this will affect accuracy (false negatives)
- Depending on the number of bugs caught, manually analyze the source code to determine whether there was an actual bug - also accuracy (false positives)
- Create a performance accuracy tradeoff graphs per consistency model. X axis = different ?time units?, Y axis = number (or percentage, etc.) of FPs and FNs (two data lines).

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., 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.

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

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

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

[12] 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).

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

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

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

[16] 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.