Symbolically Executing Real Applications

Determining the semantic equivalence of Busybox and GNU Coreutils

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^2E 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^2E 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^2E 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 software compiled from source with a pre-compiled binary could reveal malicious or accidental backdoors. In particular, the fact that S^2E runs in a VM would allow for it to be used to test kernel level programs such as file systems (or compare the read and write operations of different file systems).

We have encountered several challenges in using the S^2E platform; one in particular forced us to change the initial goal of our experiment (evaluating two consistency models from S^2E) 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 of why they failed or succeeded and what impact required changes 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 [17]. For example, Listing 1 shows an 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.

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.).

```
1 if (x == 10) {
2     // path 1
3 } else {
4     // path 2
5 }
```

Listing 1. Code example where fuzzing generally has poor coverage

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^{2}E 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^{2}E, 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 exists 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.

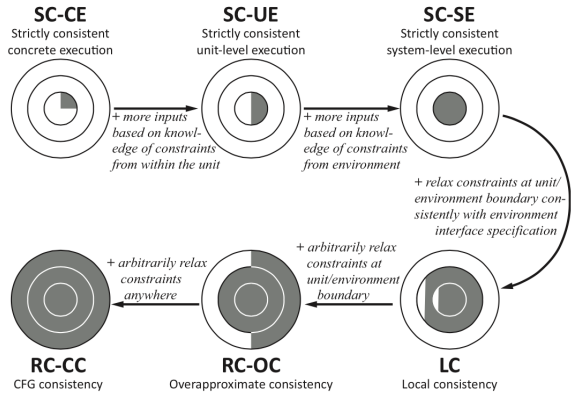

Fig. 1. S^{2}E Consistency Models [1]

- *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^{2}E and other similar symbolic execution engines. For instance, KLEE uses file system models [6] to avoid symbolically executing the actual filesystem. S^{2}E 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 consistency model similar to S^{2}E's SC-SE, but it is limited to code-based selection and one consistency model. S^{2}E 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^{2}E, which is key to S^{2}E'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^{2}E 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 S^{2}E 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 implementations of the same program using `s2e_assert`, which performs a regular assert but also does some cleaning up for S^{2}E 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. It should be mentioned that we did not elect to symbolically execute the entire system as it took too long as the QEMU VM was configured with 48 MB of RAM and attempts to increase RAM resulted in crashes.

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 as such did not provide information regarding the semantic differences between the two implementations. 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.

Listing 2 shows the command used to execute the binary of GNU Coreutils' `echo`. In order to symbolically execute the binary we must pre-load the `init_env` library (line 1) which intercepts the programs entry point invocation, parses the command line program arguments and configures symbolic execution before invoking `echo`'s main method [22]. We also have to specify some options for our symbolic execution (line 2); `--select-process-code` enables forking only in the code portion of the current binary, and `--sym-args 0 2 4` generates 0-2 symbolic arguments of length 4 for `echo`. Finally, we use `s2ecmd kill` (line 3) to kill the execution path once `echo` returns to prevent S^{2}E from running forever.

```
1 LD_PRELOAD=/path/to/guest/init_env/init_env.so \
2  /bin/echo --select-process-code --sym-args 0 \
3  2 4 ; /path/to/guest/s2ecmd/s2ecmd kill 0 \
4   "echo done"
```

Listing 2. Command for symbolically executing the GNU Coreutils `echo` binary with S^{2}E

B. Executing the binary from within a tester program

After determining that we would need to invoke both of the implementations from within the same program to compare their semantics, we elected to use the direct approach shown in Listing 3. Here we created a symbolic variable x (lines 1 and 2), created the command to invoke `echo` using `snprintf` (line 3), then enabled S^{2}E's forking (line 5) before using `popen` to execute the command (line 7). We could not use `execv` here because we would not have been able to read what `echo` outputted since `execv` forks a new process, whereas `popen` creates a pipe between the program and the command, and returns a pointer to a stream that can be read from. Later in the code (line 10), we used `fread` to retrieve that stream and read its contents into a buffer. Unfortunately, this technique did not work because S^{2}E was symbolically executing `popen` and so the analysis took too much time, consumed too much memory, and did not yield useful results.

```
1  FILE* pipe; char command[70]; char x[2];
2  s2e_make_symbolic(&x, sizeof(x), "x"); // make x
       symbolic
3  int len = snprintf(command, sizeof(command),
       "echo %s", x);
4  if (len <= sizeof(command)) {
5      s2e_enable_forking();
6      pipe = popen(command, "r");
7      s2e_disable_forking();
8  }
9  ...
10 char buf[100];
11 fread(buf, 1, 1024, pipe);
```

Listing 3. Code to symbolically invoke the GNU Coreutils `echo` binary from a tester program

C. Creating libraries for each implementation

To avoid the problem of executing the binaries from the tester program, we elected to invoke the `echo` implementations from BusyBox and GNU Coreutils directly. In order to do so, we started off with creating libraries for each implementation so that the `echo` function could be called directly from the tester program.

BusyBox: BusyBox supports creation of a shared library out of the box. Listing 4 shows the two commands required - line 1 is the most important as you set the option for Busybox to generate the shared library (and disable the generation of

the static binary) using the graphical configuration menu. This shared library was used later with the tester program, to directly call the `echo` implementation from Busybox.

```
1 make menuconfig
2 make
```

Listing 4. Commands for building GNU Coreutils as a static library

GNU Coreutils: GNU Coreutils does not support the creation of a shared library out of the box (as Busybox does) so we had to manually compile the `echo` implementation from GNU Coreutils into a library. While doing so we observed that `echo.c` was dependent on some other static libraries such as `libcoreutils.a` and `libver.a`. There are two criteria to compile a source file into a shared library: the dependent libraries must be compiled with the `-fPIC` flag from `gcc` and, if they are not, then the libraries should be for 32 bit architectures (due a limitation of `gcc`). Since `libcoreutils.a` and `libver.a` are not compiled with `-fPIC` and are compiled as 64 bit programs, we could not generate a shared library of the echo program.

Our next approach was to generate a static library that could be compiled with the tester program. Listing 5 shows the commands used to generate a working static library for GNU Coreutils after downloading the source code. Lines 1 and 2 are necessary as they generate some header files and dependent libraries that are required to build `echo` as a standalone program. Compiling echo into an object in line 4 requires options: `-g` to generate debugging symbols used by S^{2}E when using `ExecutionTracers` such as the `TestCaseGenerator`, `-I` to include the header files generated in the previous steps, `-c` to not run the linker and hence output an object file rather than a binary, and `-O2` to increase the performance of the code which is useful when using symbolic execution as the execution time is a constraint. Using the linker in line 5 requires the option `-r` to merge multiple object files and static libraries into one object file. In this step we also included all of the dependent libraries which `echo` requires. The archive step on line 6 requires the options `-q` and `-c` in order to use quick append (which adds the files to the end of the archive without checking for replacement) to speed up the operation and create an archive, respectively.

```
1 ./configure
2 make all
3 cd src
4 gcc -std=gnu99 -I../lib -O2 -g -c -o echo.o echo.c
5 ld -r -o tmpecho.o echo.o ../lib/libcoreutils.a
      libver.a
6 ar -cq libecho.a tmpecho.o
```

Listing 5. Commands for building GNU Coreutils as a static library

Using the shared libraries: We performed the aforementioned tasks, creating a shared library for Busybox and a static library for GNU Coreutils on a 64 bit Ubuntu install. We successfully built the respective libraries and compiled them with the tester program, `test.c` (provided in the Experiments section for reference), using the commands in Listing 6. The `-l` option is used in line 2 to tell `gcc` to search the library `busybox` when linking.

```
1 export LD_LIBRARY_PATH=/home/s2e/:$LD_LIBRARY_PATH
2 gcc -o test test.c libecho.a -L. -lbusybox
```

Listing 6. Commands for building GNU Coreutils as a static library

Other challenges: Next, we tried to compile the tester program on the QEMU VM that S^{2}E executes on, however it failed. This was due to the fact that the libraries were compiled on the 64 bit system and the QEMU VM was 32 bit. Initially we tried to compile directly on the QEMU VM, however, required dependencies were missing and the VM does not have internet access so we were unable to install them. We elected to install a 32 bit Debian OS in a separate virtual machine and generated the libraries there. We then succeeded in getting the tester program compiled with the libraries, and were able to execute each of the `echo` implementations from the tester program on the QEMU VM.

D. Redirecting stdout to a buffer

Once we had successfully managed to invoke the two implementations of `echo` from our tester program we needed to retrieve their outputs in order to compare their semantics. As both implementations write output directly to the shell we elected to redirect `stdout` using the commands in Listing 7. Although this worked for GNU Coreutils, the invocation of Busybox's implementation of `echo` caused our buffer to be corrupted. Upon investigation, we discovered that although GNU Coreutils uses `stdio`, Busybox avoids using it as the developers do not assume that `stdout` is actually open (and failed writes fill up the input buffer which cannot always be flushed), so they use `writev` to output the result.

```
1 char buf[128];
2 freopen("coreutils.out", "w", stdout);
3 setbuf(stdout, buf);
```

Listing 7. Code snippet for redirecting stdout to a buffer

At this point we decided to simply write the result to a file (omitting lines 1 and 3 of Listing 7), which solved our corruption issues. However, when we tried to symbolically execute the program we discovered that S^{2}E was only executing concretely! Upon further inspection it became apparent that S^{2}E concretized the value of the buffer because our symbolic program argument was only being read from and not written to. Encountering this roadblock, we elected to look into KLEE's methodology for their comparison of the two utility suites. We discovered that KLEE was symbolically modelling `stdout` (command line option `-sym-stdout`) [23]. Since S^{2}E is built upon KLEE we tried to add this flag into the `kleeArgs` portion of the configuration file used by S^{2}E, however this option did not exist, most likely due to the fact that S^{2}E is intended as an in vivo platform that uses a real software stack (i.e., it avoids modelling parts of the system). We see that if the program of interest is not reliant

on other parts of the software stack, modelling can provide a significant performance speedup without impacting the results of the analysis.

E. Modifying the implementations to not use stdout

The major challenge to get S^{2}E working with our tester program was to catch the output of the Busybox and GNU Coreutils `echo` implementations. As discussed in the above sub-section, redirecting `stdout` to a buffer did not work out as expected. We thought of modifying each implementation to return a string to the caller function instead of outputting it to the `stdout`. In the case of Busybox, it already dynamically allocated a string filled with the output values. We simply return this string object that contains the `echo`ed value to the caller function instead of outputting it. In the case of GNU Coreutils' `echo` implementation, it does not store the output value in a specific variable. Instead, wherever necessary, it uses `putc()` and `fputs()` functions to send the output value (character or a string) directly to `stdout`. We replaced each of these function invocations with the `strcat()` function, which captures the output value into a dynamic variable that we return to the caller function.

After modifying the implementations of `echo` to return a string and creating libraries for each implementation, we were able to successfully capture the output of each implementation. This appears to be an implicit requirement for any equivalence testing of programs that output their result using S^{2}E.

V. Experiments

For our experiment to check the semantic equivalence of two implementations of `echo` we used GNU Coreutils version 8.23 and Busybox version 1.22.1 (the latest versions at the time of this writing). The configuration file used to run S^{2}E is provided in Listing 8 for reference. The use of the `ExecutionTracer` on line 13 and the `TestCaseGenerator` on line 16 were necessary to output the constraints for each state and generate concrete values for failed asserts, respectively.

```
1  -- File: config.lua
2  s2e = {
3    kleeArgs = {
4      --Switch states only when the current one
           terminates
5      "--use-dfs-search"
6    }
7  }
8  plugins = {
9    -- Enable S2E custom opcodes
10   "BaseInstructions",
11
12   -- Basic tracing required for test case
         generation
13   "ExecutionTracer",
14
15   -- Enable the test case generator plugin
16   "TestCaseGenerator",
17 }
18
19 pluginsConfig = {}
```

Listing 8. S^{2}E configuration file used for experiment

Once we made changes to both implementations of `echo` to return strings rather than output the result and created shared/static libraries to invoke these functions from our equivalence checking program, we were able to run both programs symbolically and compare their results using the program in Listing 9. On lines 8-9 we allocate the input character arrays on the heap as Busybox expects them to be a dynamically allocated and the implementation attempts to free the memory. On lines 13-15, we create an input variable x, make it symbolic, and add it to the input character array before enabling forking in S^{2}E (line 17). On lines 18-19 we call the two implementations of `echo` (GNU Coreutils = `printecho`, Busybox = `printbbecho`), on line 21 we assert that the two implementations produce the same result, and on line 23 we kill the state (this code is only reached if the assert was true).

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include "s2e.h"
5
6  int main() {
7      char** str = malloc(2*sizeof(char*));
8      str[0] = malloc(2*sizeof(char));
9      str[1] = malloc(2*sizeof(char));
10     memset(str[0], 0, 2);
11     memset(str[1], 0, 2);
12
13     unsigned char x[2];
14     s2e_make_symbolic(&x, 2, "x");
15     memcpy(str[1], x, 2);
16
17     s2e_enable_forking();
18     char * out1 = printecho(2, str);
19     char * out2 = printbbecho(2, str);
20
21     s2e_assert(memcmp(out1, out2, 2)==0);
22
23     s2e_kill_state(0, "Success");
24     return 0;
25 }
```

Listing 9. Program used to perform equivalence testing on GNU Coreutils and Busybox

Our analysis produced results similar to those shown in Listing 10 which provides the constraints for that state (lines 2-7 for state 6) and also a sample test case that satisfies those constraints (lines 29-31 for state 6). Such output exists for states that fail their assertion (state 6) as well those that exit normally (state 7). There were five assertion failures in our output falling under two categories: those that ended with `0x00` (i.e., the `NULL` character), and those that included extended ASCII values like ¬ (expressed as a signed integer -0x65).

Using the sample input provided by S^{2}E we manually analyzed the two implementations of `echo` to see why there were differences. We determined that although both implementations of `echo` print a space between strings found in `argv`, the conditions leading to printing a space are different. In GNU Coreutils' implementation, if there is at least one string to echo (check `argc` $>$ `0`) a space is added after the current word. Conversely, in Busybox's

implementation a space is only printed after a string if there is another string to print after the current one (check `*++argv != NULL`).

Unfortunately we did not have sufficient time to look into the root cause of extended ASCII value mishandling because the implementations of `echo` do not discuss encodings. However, we conjecture that since both of these tools use different mechanisms to output (i.e., `stdio` vs `writev`) one of these mechanisms has not implemented support for extended ASCII.

On a personal note, we had our own experience where S^{2}E found a bug due to our own error when changing the implementation of GNU Coreutils' `echo` to return a string rather than output the result. Here we had accidentally changed the code which prints a newline after the echo is completed, with code to print a space. Luckily, S^{2}E was able to bring this error to our attention and we fixed this before running our actual experiments.

```
131 [State 6] Forking state 6 at pc = 0x804923a
    into states:
    state 6 with condition (Eq (w32 0)
    (And w32 (Add w32 (w32 4294967227)
        (Concat w32 (w8 0)
            (Concat w24 (w8 0)
                (Concat w16 (w8 0) (Read w8 1
                    v0_x_0))))) (w32 255)))
    state 7 with condition (Not (Eq (w32 0)
    (And w32 (Add w32 (w32 4294967227)
        (Concat w32 (w8 0)
            (Concat w24 (w8 0)
                (Concat w16 (w8 0) (Read w8 1
                    v0_x_0))))) (w32 255))))
...
131 [State 6] Switching from state 6 to state 7
132 [State 7] Killing state 7
132 [State 7] Terminating state 7 with message
    'State was terminated by opcode
            message: "Success"
            status: 0'
TestCaseGenerator: processTestCase of state 7 at
    address 0x8048cb9

v0_x_0: 2d -30 0
132 [State 7] Switching from state 7 to state 6
...
132 [State 6] Killing state 6
132 [State 6] Terminating state 6 with message
    'State was terminated by opcode
            message: "Assertion failed:
                memcmp(out1, out2, 2)==0"
            status: 0'
TestCaseGenerator: processTestCase of state 6 at
    address 0x8048cb9

v0_x_0: 2d -45 E
```

Listing 10. Partial output of equivalence testing for GNU Coreutils and Busybox

VI. Status

After several failed attempts, we ended up successfully performing equivalence testing on the Busybox and GNU Coreutils implementations of `echo` using S^{2}E. We have successfully used our program to symbolically execute both programs and analyzed the results from S^{2}E, as described in the Experiments section. We believe that the steps we took to perform this equivalence testing can be generalized to any GNU Coreutils/Busybox program.

VII. Conclusion

The central focus of this experiment was to understand how a complex system like S^{2}E can be applied to existing software and we wanted to document this process. To perform this experiment, we thought of comparing the semantics of two implementations of `echo` from Busybox and GNU Coreutils. We faced challenges throughout the experiment such as invoking Busybox and GNU Coreutils from our tester program. After many unsuccessful attempts, we successfully created a program which performs equivalence testing between the two implementations of `echo` using S^{2}E. We uncovered two semantic differences between each `echo` implementation: the handling of extended ASCII values and input ending with a `NULL` character. Each implementation generates different results: e.g., if the input string ends with a `NULL` character, GNU Coreutils will output a space and a newline, whereas Busybox will only output a newline.

Throughout this experiment, we learned few lessons. One of the factors which prolonged the process of getting started with S^{2}E was the lack of documentation. For any small task, we had to use the trial and error method to successfully accomplish that particular task. For instance, there is no interface for transferring files between the host (Ubuntu) and the guest (QEMU) system, and there is no documentation to perform this task. We tried using `scp`, `ftp` and even compiling our own QEMU VM, but nothing worked. Eventually, we had to post a question on the developer forum to seek assistance, which took time to get a response. If a simple task such as this was mentioned in the documentation then we would have been able to dedicate more time to actually testing programs on S^{2}E. We believe this to be one of the primary factors hindering the adoption of S^{2}E, which appears to have no adoption in industry whereas other symbolic execution platforms such as Java Pathfinder SE [21] and SAGE [17] are being used in large organizations such as NASA, Fujitsu, and Microsoft.

Additionally, we learned was how an incomplete understanding of S^{2}E's symbolic execution led to us a lot of mistakes, as we described in the implementation section. In particular, the lack of modelling for `stdout` meant that we couldn't simply capture the output, but rather had returned it as a string. Further, we identified a point where modelling parts of a system can be useful: it provides a performance tradeoff between symbolically executing an entire OS (e.g., when capturing `stdout` as a symbolic output) and simply modelling `stdout` in such a way that it can already be symbolic. Symbolic modelling provides a significant speedup for analyzing small programs such as `echo`, however programs whose correctness relies heavily on other layers in the software stack would likely not provide as valuable results.

VIII. Future Work

We would like to explore the following changes/additions to our current methodology in the future:

- Perform these tests on the rest of the utilities in Busybox and GNU Coreutils, and determine the semantic differences between them.
- Try testing other programs that don't use `stdout` and determine if they present any new challenges; if so document the process to aid future users of S^2E.
- The VM of QEMU used only 48 MB of memory, and we were unable to increase the memory (doing so led to the emulator crashing). As such, figuring out a way to change QEMU launch parameters for S^2E in order to give it more memory would be crucial to using S^2E to symbolically execute larger programs or the entire software stack.

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.

[21] ANAND, S., PASAREANU, C. S., AND VISSER, W. JPF-SE: A symbolic execution extension to Java Pathfinder. In Proc. of the 13th International TACAS Conference, 2007.

[22] How to symbolically execute Linux binaries?, S^2E. Jan 2013. `https://github.com/dslab-epfl/s2e/blob/master/docs/Howtos/init_env.rst` (Accessed: November 2014)

[23] Testing Coreutils, KLEE. 2009. `http://klee.github.io/tutorials/testing-coreutils/` (Accessed: December 2014)