Carving Parameterized Unit Tests

Alexander Kampmann
CISPA Helmholtz Center for Information Security
Saarland Informatics Campus, Saarbrücken, Germany
alexander.kampmann@cispa.saarland

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarland Informatics Campus, Saarbrücken, Germany
zeller@cispa.saarland

***Abstract*—We present a method to automatically extract (“carve”) parameterized unit tests from system test executions. The unit tests execute the same functions as the system tests they are carved from, but can do so much faster as they call functions directly; furthermore, being parameterized, they can execute the functions with a large variety of randomly selected input values. If a unit-level test fails, we lift it to the system level to ensure the failure can be reproduced there. Our method thus allows to focus testing efforts on selected modules while still avoiding false alarms: In our experiments, running parameterized unit tests for individual functions was, on average, 30 times faster than running the system tests they were carved from.**

***Index Terms*—fuzzing, carving, system tests, unit tests**

I. Introduction

In this paper, we present a method that joins the benefits of both system-level and unit-level test generation. A *unit test* tests a single unit of a program, e.g. an individual function, allowing for effectively narrowing down the scope of analysis and execution. The downside, however, is that synthesized function calls may *violate implicit preconditions*: If a test generator finds that `sqrt(-1)` crashes, this does not help developers who never intended `sqrt()` to work with negative numbers anyway. When generating tests at the *system level,* this problem of false failures does not occur, as any failure caused by third-party system input needs to be fixed. On the other hand, system-level testing generates lots of *overhead* as input must be read, decomposed, processed, and more before a function of interest is finally reached.

Our key idea is sketched in Figure 1. It is based on the concept of *carving unit tests* [1], observing system executions to extract unit tests that replay the previously observed function executions. However, we extend the concept by extracting *parameterized unit tests*. To this end, we identify those function arguments that are directly derived from system input. These arguments then become unit tests parameters, allowing for extensive fuzzing with random values. We can thus random test individual functions with hundreds of values, with all invocations in the context of the original run. Afterwards, we *lift* our unit-level findings back to the system level, to verify that they are real failures.

II. Approach

A. Carving Unit Tests

Carving [1] is accomplished by recording all parameter values for a function invocation during execution. Then, a unit test which invokes individual functions with the same parameters is generated.

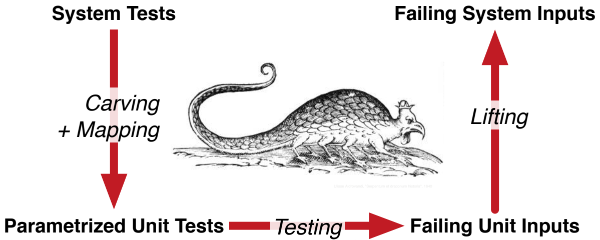

Fig. 1. Overview of our approach.

1) Example: As an example of a carved unit test, consider the function `bc_add()` from the `bc` calculator program.

```
void bc_add (bc_num n1, bc_num n2, bc_num *r, int scale_min);
```

`bc_add()` accepts two numbers, `n1` and `n2`, and writes the sum of those two numbers to the number pointed to by `r`. `scale_min` gives the minimal number of floating-point positions to be used by `result`.

From an execution of `bc` with a concrete input (say, `"1 + 2"`), our Basilisk implementation observes the call `bc_add(1, 2, &result, 0)`. Thus, it creates the unit test:

```
void test_bc_add() {
    // set up the context
    bc_num n1, n2, result;
    bc_int2num(&n1, 1); bc_int2num(&n2, 2);
    // call the function under test
    bc_add(n1, n2, &result, 0);
}
```

2) Implementation: Our Basilisk prototype implements carving based on the *low-level virtual machine* (LLVM) [2]. LLVM provides an intermediate representation (LLVM IR), which was designed for static analysis.

Basilisk works in two phases. It statically instruments the LLVM IR code, inserting probes which report all method invocations including the parameters, as well as all writes to global variables. During execution, those probes write the observed values at those points to a trace file.

For primitive types, like ints or floats, observed values can be written directly. For structs, we recursively dump their members. For pointers, we intercept calls to `malloc` to be able to recognize how large the pointed-to area is.

TABLE I
Branch coverage achieved by Basilisk and Radamsa

			#System Tests		Coverage	
Subject		**LoC**	**Basilisk**	**Radamsa**	**Basilisk**	**Radamsa**
b2sum	checksum calculation	115	358.0	629.0	37.93%	19.49%
paste	text processor	79	280.0	346.3	33.33%	31.08%
tac	text processor	111	89.6	212.6	34.66%	30.71%
bc	arbitrary-precision calculator	151	169.0	577.2	26.47%	28.46%
dc	arbitrary-precision calculator	136	135.4	434.6	18.39%	41.06%
cut	text processor	127	339.2	3117.1	21.00%	20.50%
sed	text processor	215	175.7	1058.3	21.19%	15.73%

B. Parameterizing Carved Unit Tests

In parameterization, we identify *parameters*, values which can later be set by the fuzzer. We therefore restrict ourselves to *values that are derived directly from system-level input.*

To match variables and their origins in the system input, we use a simple, yet efficient approximation. For each value v, we check:

- If v is a string, we check whether it is a substring of the system input.
- If v is numeric, an integer or a floating-point number, we check whether the decimal representation is a substring of the system input.

If we find a match, we mark v as a parameter. Instead of using the recorded value v, we now allow the fuzzer to insert a new value v' into the unit test, as a replacement for v.

1) Example: In the running example, Basilisk identifies `1` and `2` as coming from system input, and thus makes them parameters of the carved parameterized unit test `test_bc_add(p1, p2)`:

```
void test_bc_add(int p1, int p2) {
        // set up the context
        bc_num n1, n2, result;
        bc_int2num(&n1, p1); bc_int2num(&n2, p2);
        // call the function under test
        bc_add(n1, n2, &result, 0);
}
```

C. Fuzzing Function Calls

Once we have a parameterized unit test, we can use a fuzzer to choose new values for the parameters. A fuzzer basically provides random values.

1) Example: In the running example, we can now invoke `bc_add()` with random values for `p1` and `p2`:

```
test_bc_add(337747944, 352295539);
test_bc_add(535612873, 790525737);
// ... and more
```

D. Lifting

Out of the generated unit tests, we select some for *lifting*. A test is selected for lifting if it either causes a unit-level crash, or yields new unit-level coverage.

In section II-B, we parameterized values if they occur as part of the system input. Now, we replace those parts of the system inputs with the new values that were found by the fuzzer. This yields a new set of system-level inputs.

If those system-level inputs trigger the same behaviour as the unit test, a failure or new coverage, depending on why the test was selected for lifting, we report it to the developer. This leads to a zero false-positive rate.

1) Example: In our example, let us assume that `test_bc_add(10, 20)` fails. From the original run, we know that the arguments `p1` and `p2` correspond to the values `1` and `2` in the system input. In the input, we would thus replace the values `1` and `2` by the failure-inducing values of `p1` and `p2`, resulting in the input `10 + 20`. Only if `bc` fails on this input would we report the failure.

III. Evaluation

Our unit tests are on average 30x faster than system tests generated by Radamsa [3]. This means that we can test more inputs in less time, yielding a higher probability to trigger new behaviour.

This manifests in four out of seven programs, as listed in table I. For four programs, `b2sum`, `paste`, `tac` and `sed`, we achieved better coverage. For `cut`, the result is not significant.

For all programs, Basilisk generates a lower number of tests in total. The reason is that Basilisk lifts only (unit-level) test inputs which trigger new behavior, and thereby generated system-level tests are more likely to trigger new behavior than those generated by Radamsa.

IV. Conclusion

Carved parameterized unit tests, like system tests, provide a realistic context for test generation, like unit tests, they can be analyzed and executed quickly. When a unit fails, the failure can be lifted to the system level and validated there, either suppressing a false alarm or yielding a failing system test. This means that our technique provides less false positives than unit-level test generation, but maintains it's advantages in speed.

References

[1] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, “Carving and replaying differential unit test cases from system test cases,” *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2009.

[2] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, Mar 2004.

[3] A. Helin. Radamsa. [Online]. Available: https://gitlab.com/akihe/radamsa