Software Verification for C using Serval

Chinmay Garg
chinmay@ucsb.edu

Brian Lim
blim@ucsb.edu

Aneesha Mathur
aneeshamathur@ucsb.edu

Abstract

Software verification is a tedious and time consuming task when done properly. This challenge was taken by the authors of `Serval` [1], where they introduced a new way to perform an almost automated push-button verification. This framework uses Rosette for symbolic execution to check for bugs. However, the authors only check for certain bugs such as buffer overflow or oversized shifting. We created additional tests that check for integer overflow, division by zero, illegal memory access and synchronization issues. We also take a look into and understand why Serval is not able to handle infinite loops. Throughout this process, we are able to identify and understand the advantages and limitations of Serval in program verification.

1 Introduction

Through the years, formal verification has proven to be important in eliminating entire classes of bugs, but it comes at a price. It took many person-years to write formal proofs and construct a machine-checkable proof to show that the implementation satisfies the specification. But recently, it has achieved a higher degree of automation through more recent push-button approaches. Despite that, these systems are vulnerable to becoming very specific to targeted systems and cannot be reused easily.

Torlak's paper talks about their automated verifier (Serval) created for `RISC-V, x86-32, LLVM` and `Berkeley Packet Filter (BPF)` which is tested (retrofitted) on two security monitor systems - `CertKos` and `Komodo` and compares the verification results produced (and the bugs discovered) by Serval against the verification originally produced for these systems by Coq and Dafny.

An important feature of Serval is that we can write our application code in a language of our choice (for the purpose of this project we have used C) and compile it using the standard toolchain (which is LLVM in our case). Additionally, we also need to write the specifications of the desired system behavior in Rosette. Serval provides a library to simplify the task of writing specifications, including state-machine refinement and non-interference properties.

Serval leverages Rosette to write an interpreter for an instruction set which can transform a regular program to work on symbolic values. The specifications and implementations are encoded into symbolic values and Serval then employs Rosette to produce SMT constraints. It invokes a solver to check the satisfiability of the produced constraints for verification. The developers can also add the functionality that in case of verification failure, the solver generates a counter example and that is given as an output to help the developer better understand the cases which cause the bug. Verification failures can happen either due to insufficiently written specification or an erroneous implementation.

Through our project, we make use of the various examples shared by the authors of Serval in understanding the functionality and write several code snippets in C to verify whether Serval is successfully able to identify and detect the bugs to check the validity of the code.

2 Methodology

In this section we will go over the methodology we used in developing our tests using the Serval framework. We will talk about details of the individual tests within Section 3.

2.1 Overview

Initially, to understand the existing functionality of this tool, we used the Serval SOSP'19 tutorial provided by the authors to better understand state machine refinement and safety specifications for a Toy Monitor kernel.

We also contacted the authors for a better understanding of their file structure and started by recreating and understanding some of the tests they had already written as a part of their framework distribution. This gave us deep insight into how Serval functions and the kind of input it is expecting for it to be able to understand our C code.

```
if (T->isIntegerTy()) {
        OS << "(bitvector " << T->getIntegerBitWidth() << ")";
} else if (auto ST = llvm::dyn_cast<llvm::StructType>(T)) {
    /* We should never print out struct type names. */
    assert(ST->isLiteral());
    /* LLVM prints { ... } for literal struct types; change to (list ...). */
    OS << "(list";
    for (auto ET : ST->elements())
        OS << " " << getType(ET);
    OS << ")";
} else if (auto VT = llvm::dyn_cast<llvm::VectorType>(T)) {
    /* We don't really support vector types, just to avoid syntax errors. */
    OS << "(make-vector " << VT->getNumElements()
       << " " << getType(VT->getElementType()) << ")";
} else {
    /* unsupported type */
    OS << '"' << *T << '"';
}
```

Listing 1: Rosette Emitter for lists - Serval Framework

2.2 Execution

Serval essentially works as a interpreter between the C code and Rosette test implementation. The C code provides a baseline for the Serval framework which then generates the `llvm` instruction code using `--emit-llvm`. This file is then parsed using the Rosette Emitter as seen in Listing 1. The code here takes each symbol from the `llvm` instruction code and parses it word by word to generate a `list()`. They also have parsing for unsupported types in order

to avoid errors. Since they do not currently support C++ any datatype associated with such is not supported as well.

```
(define (@addition %a %b)
; %entry
  (define-label (%entry) #:merge #f
    (set! %add (add %b %a))
    (ret %add))

  (define-value %add)
  (enter! %entry))

(define (@addition_overflow %a %b)
; %entry
  (define-label (%entry) #:merge #f
    (set! %add (add %b %a))
    (ret %add))

  (define-value %add)
  (enter! %entry))
```

Listing 2: Generated Rosette Code from C file - Rosette Test

Once this is converted into a Rosette code, this can now be utilized within our tests to call to and verify the program. An example of the generated Rosette code can be seen in Listing 2. Once the tests and specification of the program has been written in the test code, we can run these tests using `raco test <test>.rkt`.

3 Implementation

During this project all the tests were run on the docker container provided by the authors of the Serval paper. All the tests were lightweight tests thus did not require an extended period of time to check so the container specifications are not important here.

Within our project scope, we tested for bugs such as Integer Overflow, Illegal Memory Access, Division by Zero and Mutex locks. Almost all of the tests were simple and easy to write which is what Serval planned to accomplish and cut down on program verification time from months to couple hours. This is possible due to the conversion of C code into Rosette code using the `llvm` interpreter written in C++ by the authors of Serval as a part of their framework. We evaluate these results by proving the bugs exist using the Serval framework.

3.1 Integer Overflow

Integer Overflow is a common yet an high priority bug in most software since it can and has caused major issues in the past. We take a look into this using the simple C implementation of adding two integers is as given in Listing 3.
The associated racket file contains two functions: one which verifies the functionality of the C code and the other which employs Rosette to verify if the written C code is bug-free.

```c
int32_t addition(uint32_t a, uint32_t b) {
  return (short)a+b;
}
```

Listing 3: Integer Overflow - C Code

```
(define-symbolic x y i32)

(define (check-symbolic-addition)
  (check-equal? (@addition x (bv 0 i32)) x)
  (check-equal? (@addition x (bv 1 i32)) (bvadd x (bv 1 i32)))
  (check-equal? (@addition x y) (bvadd x y))

(define (check-buggy-addition)
  (define sol (verify (@addition x y)))
  (check-unsat? sol))
```

Listing 4: Integer Overflow - Rosette Test

The `check-symbolic-addition` method only verifies the functionality of the C method - addition. We compare the results of the the Rosette-equivalent @addition function and `bvadd`, which is described in the Rosette language. The next method check-buggy-addition utilizes the solver-aided programming concepts of Rosette. We use verifier to compute the satisfiability of the C-code addition. The check-unsat is merely a syntactic sugar and uses `sat?` under the hood to get the results. We are able to detect that the code is `unsat`, that proves Serval is able to detect this bug.

3.2 Division by Zero

```c
uint32_t division(uint32_t x, uint32_t y) {
  if (y != 0)
          return x / y;
  return 0;
}

uint32_t division_bug(uint32_t x, uint32_t y) {
  return x/y;
}
```

Listing 5: Division by Zero - C Code

Looking at the C implementation for division by zero in Listing 5, we can notice that there are two version of the division function. One is where it makes sure that the denominator is not 0 and will not cause an issue and the other one is a more risky division bug where there is no check.

Serval takes this C implementation and generates the Rosette code for that same implementation. Now we develop a spec for the program as seen in `check-div`, that now mirrors the same

```
#lang rosette

(define-symbolic x y i32)

(define (check-div x y #:result [result (/ x y)])
  (check-equal? (@division (bv x i32) (bv y i32)) (bv result i32)))

(define (check-symbolic-division)
  (check-equal? (@division x (bv 0 i32)) (bv 0 i32))
  (check-equal? (@division x (bv 1 i32)) x)
  (check-equal? (@division (bv 0 i32) y) (bv 0 i32))
  (check-equal? (asserts) null))

(define (check-buggy-div)
  (define asserted
    (with-asserts-only
      (@division_bug x y)))
  (check-equal? (asserts) null)
  (check-equal? (length asserted) 1)
  (define cond (first asserted))
  (define sol (verify (asserted cond)))
  (check-sat? sol)
  (evaluate cond sol))
```

Listing 6: Division by Zero - Rosette Test

behavior that we want out division to accomplish. Now the test `check-symbolic-division` calls the bug free version of division and checks if the result matches the one we expect. Similarly it checks the `check-buggy-div` on symbolic values of `x` and `y`. Since, the values are any integer32 values they can be 0 and thus it will fail the division returning that the solution wasn't satisfiable. Serval is able to successfully detect this bug.

3.3 Illegal Memory Access

Illegal memory accesses are fairly common and can cause `SIGSEGV` crashes that can cause serious issues within applications. For our test we have a C implementation that defines the values for the `current_user` using the registers that are passed from within the Rosette code. A very simple code snippet of importance can be seen in Listing 7.

```
long illegal_memory_1(void) {
    return dictionary[current_user+1];
}
```

Listing 7: Illegal Memory Access - C Code

When we call this program using values of `current_user` that max out at the value we have set as `MAX_USERS` it will fail the specification and return a incorrect memory access value for which Serval is successfully able to print a counter example. A similar test was a part of one of

the simple tests within the Serval SOSP'19 Tutorial. This test was adapted and written based on that to check the bug.

3.4 Mutex Locks

```
#lang rosette

(define (check-global-val x)
  (parameterize ([current-machine (make-machine symbols globals)])
    (define exp (bv x i32))
    (check-not-equal? (@get_count_value) exp)
    (@set_count_value exp)
    (define act (@get_count_value))
    (check-equal? act exp)
    (check-not-equal? (@get_lock_value) exp)
    (@set_lock_value exp)
    (define lock_act (@get_lock_value))
    (check-equal? lock_act exp)))

(define (check-global-concrete)
  (check-global-val 1)
  (check-global-val 0)
  (check-equal? (asserts) null))

(define (impl-inv)
  (parameterize ([current-machine (make-machine symbols globals)])
  (define-symbolic x i32)
  (@set_lock_value (bv 0 i32))
  (@set_count_value x)))

(define (check-mutex)
  (parameterize ([current-machine (make-machine symbols globals)])
    (define pre-inv (impl-inv))
    (define-symbolic* x i32)
    (choose* (@increment_count x) (@get_count))
    (define inc-thread (thread (lambda() (@increment_count x))))
    (define get-thread (thread (lambda() (@get_count))))
    (for-each thread-wait (list inc-thread get-thread))))
```

Listing 8: Code snippet from mutex.rkt

For mutex locks writing tests was difficult. This was due to the fact that Serval's `llvm` framework does not support stdlib, and limits the libraries we can use including no support for `pthread, mutex` etc. Since we were not able to create threads within the C code we decided to use the Racket angelic library to perform wait using `thread-wait`. This is done in part by using `chose*` to non-deterministically select the thread we want to run. We manually lock using a `int` value to similar a mutex lock.

Due to the nature of a `wait` call it is not possible to completely verify a non-deterministic

program using Serval. In fact, all of our tests with a `while(1)` were inconclusive as the Rosette execution engine would keep the code running never reaching a `un-sat` decision. Thus we have to make an assumption that if the program checks for the current value stored in the global map and it turns out to be 1 then the lock has not been released. This is a naive test for mutex locks and exclusion and cannot be relied upon in real-world scenarios. All the code in these tests is available on the github repo [2].

4 Challenges & Observations

Initially, the biggest challenge we had was getting used to the syntactic sugar used by Rosette. This was a uphill battle as none of us had any formal experience with any functional language. Rosette offers a lot in terms of it's libraries and the functions it provides to emulate the multi-threaded nature of the C programs.

The functionality supported by Rosette is overly important as Serval's framework heavily relies on it to be able to test the functionality and run specifications as intended as Rosette code. But since it's a higher level language and the whole purpose was to be able to test the functionality of the C code, it brings us to a couple important limitations. First that Rosette cannot deal with simulation of heap, concurrency and/or parallelization well. This means that there is a whole area of possible bugs we cannot test for correctly without making strong assumptions that might not hold up with real world applications.

Another important challenge was the lack of debugging. Short of Dr. Racket there is no good debugger available for Rosette/Racket code at the moment. This means if there is something semantically or logically incorrect in one of our specifications it is hard to notice until we start getting results we don't expect. Most of the debugging within this project was done using the good old `displayln` statements.

Serval's authors state clearly in their paper that they are unable to handle non-terminating code and thus most loops not deterministically finite are not understood by the framework. But instead of filtering and crashing prior to running the specification, Serval currently performs a line-by-line conversion of the `llvm` instruction set to generate a Rosette file that contains the implementation of the C code used to be called within the testing files. They do not yet support `stdlib` which means that any feature relying on that such as `malloc, free, printf, scanf, putc, pthread, mutex` etc. cannot be used within the C code. This makes most of the real-world code difficult to quickly port to Serval for verification.

5 Conclusion

Program verification has proven to be a very important step in concrete software development process and is widely used in applications and high risk industries. As a part of this project, we wrote tests that would verify some of the common bugs that occur in C that are easy to go undetected. We do so by writing the C code with the bug introduced and a Rosette spec that we can compare it against to prove if the program is completely satisfiable or not.

We have successfully taken the Serval framework as provided by the authors to verify more common C bugs. We were able to understand and argue about the benefits of Serval along with the limitations that the current state of program verification faces. Since the Serval framework is unable to reason about non-deterministic loops it is not possible to verify parallel processing applications or concurrency. This is also in part limited by the missing `stdlib`

support. This causes any major heap functionality or concurrency libraries to not work along with standard in and out which could be a source of a lot of common binary exploits.

This was a good learning experience that helped us understand and get experience with Serval as well as Rosette in the process. It's impressive to see how easily program verification can avoid bugs that could prove to be fatal in some cases. However, the current state of program verification is far from a simple plug-n-play and there may need to be a lot of advancement before it can be utilized for real world applications without major modifications and assumptions.

References

[1] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: ACM, 2019, pp. 225–242. [Online]. Available: http://doi.acm.org/10.1145/3341301.3359641

[2] C. Garg, A. Mathur, and B. Lim, "Cs292c-project-repo," 2019. [Online]. Available: https://github.com/chippermist/cs292c-project

[3] Unsat, "Serval sosp'19 artifact." [Online]. Available: https://unsat.cs.washington.edu/projects/serval/sosp19-artifact.html

[4] Uw-Unsat, "uw-unsat/serval-tutorial-sosp19," Oct 2019. [Online]. Available: https://github.com/uw-unsat/serval-tutorial-sosp19/tree/master/serval

[5] "Racket documentation." [Online]. Available: https://docs.racket-lang.org/

[6] "Using synchronous channels for communication between threads." [Online]. Available: https://rosettacode.org/wiki/Synchronous_concurrency?fbclid=IwAR0WPVi5-64rtMZ6QfyJD3o7WRz4WsrFUyj8RvGCISGwRsBgcvYrKTgAOTU#Racket