# Poster: Verification of Network Software using Symbolic Execution

Robert Esswein*
*Electrical and Computer Engineering*
*University of Pittsburgh*
Pittsburgh, PA
robert.esswein@pitt.edu

Juliana Furgala
*MIT Lincoln Laboratory*
Lexington, MA
juliana.furgala@ll.mit.edu

Samuel Jero
*MIT Lincoln Laboratory*
Lexington, MA
samuel.jero@ll.mit.edu

***Abstract*—Software verification is a useful tool for demonstrating the absence of bugs in software. Unlike other techniques, verification methods, such as symbolic execution, allow developers to consider all possible cases of input that their software could face, providing a comprehensive proof of proper behavior. However, software verification has historically been regarded as too difficult or unnecessary for many applications. Recently, innovations in software verification tools have allowed developers inexperienced with verification to utilize these tools in their own projects. In this work, we present our experience using Kani, a symbolic execution engine for Rust, to verify portions of the smoltcp network stack. We have verified some of the lower layers of the network stack, specifically Ethernet and IP protocols and parts of UDP, and commonly used storage and time keeping structures used by many layers. In this work, we found 60 bugs in smoltcp as a demonstration of how effective verification can be in embedded systems. This work can continue to be expanded to more protocols and higher layers of the network stack.**

## I. Introduction

Faulty software can have catastrophic consequences, but bugs can make their way into software as frequently as every forty lines [1]. In the simplest case, these bugs can be introduced by simple programming mistakes. However, the complexities of real hardware and software can introduce faulty behavior that is difficult for humans to detect. The most common bugs are memory bugs [2], where a program uses memory in a way that violates assumptions of the program, such as by writing to a location that it does not have access to or failing to free unused memory. As software developers, it can be difficult to catch every single potential erroneous behavior given the scale of modern software systems. Nearly every piece of software written has some amount of untested code, allowing unexpected behavior to sneak into the codebase. This is especially a problem for highly critical, real-time software, on which mission success depends.

A number of approaches can be used to identify bugs prior to releasing a software system, the most common being testing. Writing representative unit tests, while a useful tool to demonstrate proper opertation of software, does not provide full coverage of all possible inputs. Because unit testing software relies on more software development, many common bugs can pass tests because the developers have simply not thought of them. Similar to unit testing is fuzzing, which tests randomly generated test cases. However, this still cannot cover all possible combinations of inputs that a piece of software could see. If a particular bug is only triggered by a very small part of the input-space, then statistical odds of detecting such a bug are low.

Another approach is verification. The goal of verification is to prove the correct behavior of a piece of software. Some examples of verification methods include theorem proving, abstract interpretation, and symbolic execution. Using theorem proving techniques, a developer can mathematically prove that the desired result of a program is a consequence of the program itself, leaving no room for unexpected behavior. Abstract interpretation reasons about the semantics of a program without actually running the program to gain insight on what could potentially cause faulty behhavior.

In this work, we utilized symbolic execution, so that will be the focus of the remainder of this work. In symbolic execution, the inputs to a piece of software are treated as symbolic values. Then, the symbolic execution engine runs through the program and determines how the symbolic values change the state of the variables in the software. Potential bugs in the software can be detected as violations to certain constraints in the variable state. This means that only bugs that exist within a program's state-space (not side channel or logical bugs) can be detected by symbolic execution. For example, any violation of data type bounds can be detected by ensuring that there is no possible value for the symbolic variables that will result in the bounds of a variable being exceeded. We chose symbolic execution because it is simpler to use than theorem proving, but can still accurately represent the language, which abstract interpretation struggles with.

## II. Work Completed

The goal of this work is to assess symbolic execution as a tool for checking commercially developed or open-source software. To do this, we selected the task of verifying the open-source smoltcp network stack [3] as a case-study. This is

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Air Force.

*This work was completed while working at MIT Lincoln Laboratory.

| Overflow | Division by 0 | Memory | Panic |
|---|---|---|---|
| 29 | 2 | 26 | 3 |

TABLE I
DIFFERENT TYPES OF BUGS FOUND USING VERIFICATION.

a desireable target for verification because it was designed for bare-metal, real-time systems, making it ideal for embedded or IoT applications [4], [5]. We used the Kani verifier [6], which takes Rust Mid-level Intermediate Representation (MIR), converts that to Goto-C, then uses the C Bounded Model Checker (CBMC) [7] to run symbolic execution over the code. The CBMC is a desirable tool for this job because it has a long legacy of use in many different verification projects [8], [9], [10]. Our goal is not to prove compliance to a specification, but prove the absence of certain bugs. In particular, we wanted to verify that the network stack would not cause any out-of-bounds memory accesses, arithmetic overflows, or other panics. With this in mind, we developed verification harnesses, which are functions that symbolically test a portion of the code, for this software to assess the feasibility of using software verification for other projects.

Of the smoltcp software, we focused on the lower layers of the network stack and storage structures used by multiple layers. Specifically, Ethernet and IP layers of the stack have been verified, with some higher layer protocols (such as UDP) having incomplete verification. So far, we have focused on writing verification harnesses, with corrections to code pushed to future work. In total, 3227 lines of code have been covered by verification harnesses, with 60 bugs detected. The specific breakdown of what types of bugs were found can be seen in Table I. This took 2551 lines of code dedicated to verification, and a total of 120,834.97 seconds (33 hours, 33 minutes, and 54.97 seconds) to verify.

The most interesting finding from this work was in the calculation of checksums for IP packets. Because the checksum loops over all bytes in a packet, this is a very difficult process to verify with symbolic execution due to the large number of operations with symbolic variables. Loops must be unrolled for a symbolic execution engine to reason about, meaning that looping through an entire packet is very difficult for a symbolic execution engine. We estimate that Kani would run up to 150 days to check the original implementation of the checksum calculation.

So, we explored an alternative approach. The function that performed the checksum calculation was converted into six functions that could be verified separately. The first function handles setup and final computations in the checksums. The next three functions are loops that only run sixteen times each, calling the next function inside the loop. These functions are able to calculate checksums of increasingly large packets, with each function being used by the previous. The final two functions handle checksums for either exactly thirty-two bytes or less than thirty-two bytes, respectively, with the first one able to utilize vector operations. In addition to checking each of these functions for the previously listed bugs, these functions were also verified to have a maximum possible return value, enabling them to be stubbed out of other functions. With this method of reducing loops, verification time was reduced to a mere five minutes.

## III. LESSONS LEARNED/CONCLUSION

Kani has provided a simple interface to verify code using symbolic execution. Installation and the commandline interface are both simple, and the additional code required for verification is intuitive. Kani is especially useful for trying to detect anything that would cause a runtime panic because these checks are implicitly done, while explicit user assertions require some additional development. The main drawbacks of Kani is its overhead. For the amount of code verified, nearly the same amount of code had to be written. Additionally, the amount of time required to run this verification is significantly longer than how long testing would take. So, for highly critical software, verification is worthwhile, but the additional development time required is a serious consideration for any project that uses Kani.

In this poster, we present our experience using Kani to verify Rust code, specifically the smoltcp network stack. We have found that simple verification techniques can be utilized for commercial or open-source projects, though it might require modification to how long loops are coded to make verification feasible. Future work on this project includes continuing to develop verification harnesses for the smoltcp stack and to begin the process of integrating many small patches into the codebase to pass verification.

## REFERENCES


[1] Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press.

[2] MSRC Team, "A Proactive Approach to More Secure Code," 2019, Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. [Accessed Apr. 2, 2024].

[3] smoltcp. Available online: https://github.com/smoltcp-rs/smoltcp/ (accessed on 19 July 2023).

[4] Lankes, Stefan, et al. "RustyHermit: a scalable, rust-based virtual execution environment." High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers 35. Springer International Publishing, 2020.

[5] E. Wen and G. Weber, "Wasmachine: Bring IoT up to Speed with A WebAssembly OS," 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), Austin, TX, USA, 2020, pp. 1-4.

[6] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. ACM, New York, NY, USA, 321–330.

[7] Clarke, Edmund, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs." Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004.

[8] Dörre, Felix, and Vladimir Klebanov. "Practical detection of entropy loss in pseudo-random number generators." Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016.

[9] Keerthi K., Chester Rebeiro, and Aritra Hazra. 2018. An Algorithmic Approach to Formally Verify an ECC Library. ACM Trans. Des. Autom. Electron. Syst. 23, 5, Article 63 (September 2018), 26 pages.

[10] R. Metta, "Verifying Code and Its Optimizations: An Experience Report," 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 578-583.

# Verification of Network Software using Symbolic Execution

Robert Esswein[1]*, Juliana Furgala[2], Samuel Jero[2]
[1]University of Pittsburgh, Electrical and Computer Engineering, [2]MIT Lincoln Laboratory
robert.esswein@pitt.edu, {juliana.furgala, samuel.jero}@ll.mit.edu

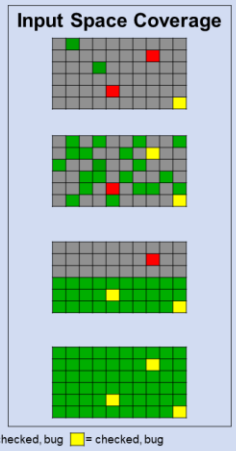
## Introduction

- Causes of bugs


- How can software developers ensure that software is free of potential bugs in such critical systems?

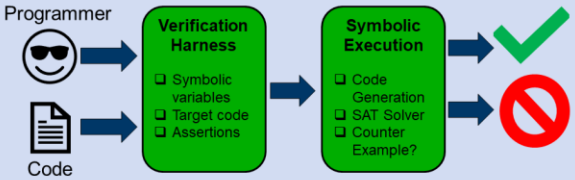
## Bug Prevention Approaches

- **Testing**
- Easy to implement
- Demonstrates correct behavior
- **Fuzzing**
- Generate test cases randomly
- Unable to find all edge cases
- **Safe Programming Practices**
- Built-in safety features
- Avoid common bugs
- **Verification**
- Demonstrate absence of bugs
- Consider all possible inputs




## Symbolic Execution

- Represent variables as symbolic values
- Iterate through program, consider possible state-space
- Search for constraint violations (e.g. variable limits)




## Approach

- Use Kani[1] to verify network code in smoltcp[2] repository
- **Kani**
  - Convert Rust MIR to Goto-C
  - Run Goto-C code in C Bounded Model Checker (CBMC)[3]
- **Goal**: Verify absence of out-of-bounds access to memory, arithmetic overflows, and other panics
- **Challenges**
  - Low level code is difficult to reason about
  - Verification can take a long time, what is worthwhile?
  - Writing harnesses that capture all possible inputs

## Verification Harnesses

```
pub struct Instant {
    micros: i64,
}
Impl Instant {
    pub fn from_millis(ms: i64) -> Instant {
        Instant {
            micros: ms * 1000,
        }
    }
}
#[kani::proof]
fn kani_time_instant_from_micros_const() {
    let i: i64 = kani::any();
    let inst = Instant::from_millis(i);
    assert!(i * 1000 == inst.micros);
}
```

Code being Verified

Verification Harness

```
#> cargo kani
```

Arithmetic Overflow

Symbolic Variable

- Harnesses allow verification of pieces of programs
- Typically, one harness verifies to one function or struct
- **Stubbing** - Replace a function for verification purposes
  - Useful if a function takes a long time to verify, but is called by another function
  - Used to verify IP checksum

## Verification Overhead

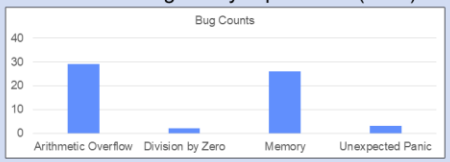
- **Steps**: symbolic variable creation, function calls, assertions
- Following these steps, many harnesses are very simple

| | |
|---|---|
| Lines of Code Verified | 3227 |
| Lines of Code Written for Verification | 2551 |
| Verification Harnesses | 149 |
| Verification time | 120,834.97 s (33 h, 33 m, 54.97 s) |

- Symbolic execution uses a SAT solver, so verification takes much longer than testing

## Results

- Verified commonly used structures (time keeping, data storage) and lower layer protocols (Ethernet, IP)
- Minimal verification for some higher layer protocols (UDP)




## IP Checksum

- Checksum calculation loops over entire IP packets
- Kani must unroll loop → that this large loop is difficult to reason about
- Estimated verification time: 150 days

**Old**

```
pub fn data(mut d: &[u8]) -> u16 {
  let mut accum = 0;

  const CHUNK: usize = 32;
  while d.len() >= CHUNK {
    let mut x = &d[..CHUNK];
    while x.len() >= 2 {
      accum += read_u16(x) as u32;
      x = &x[2..];
    }
    d = &d[CHUNK_SIZE..];
  }
}
```

**New**

```
pub fn outer(d: &[u8]) -> (u32, usize) {
  let mut accum = 0;
  let mut ind = 0;

  const CHUNK: usize = 8192;
  while ind <= d.len() - 2 {
    let mut max_i = ind+CHUNK;
    let (res, inc) = mid(&d[ind..max_i]);
    accum += res;
    ind += inc;
  }
  (accum, ind)
}
```

- Solution: split function into six smaller functions and verify independently
- Each function loops only 16 times
- First calculates checksum of 16*2 octets, second for 16*16*2 octets, etc.
- Verification time reduced to 5 minutes total
- Requires additional verification of functions' return

## Conclusions

- Demonstration of verification tools in open-source software
- Comparison of number of different types of bugs
- Measurement of the amount of effort (in lines of code) must be used to enable verification
- Insight into how code must be adjusted in order to facilitate verification

## References

1. Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in rust. In Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22).
2. M-Labs. smoltcp: smol TCP/IP stack in Rust, 2016. https://github.com/m-labs/smoltcp.
3. Clarke, Edmund, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs." Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004.

*This work was completed while working at MIT Lincoln Laboratory.



LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY