Analyzing Snort

Puneet Mehrotra*
University of British Columbia
puneet89@cs.ubc.ca

Swati Goswami*
University of British Columbia
sggoswam@cs.ubc.ca

1. ABSTRACT

Verifying systems code is hard because of concurrency and inherent complexity of the code. Understanding the invariants and other behavioral and performance characteristics of a distributed systems is challenging because multiple nodes communicate and coordinate together towards the task of performing a computation.

Verification methods and tools implicitly assume that the code to be verified is deterministic. That is users of the software will provide relevant inputs, and by verifying program behavior using all possible values of that input, a given verification methodology can draw inferences about a system.

In the case of distributed software, the assumption of implicit determinism is void. Network software typically works in collaboration with other compute nodes in order to fulfill the required functionality. For example, in a peer-to-peer system, peers detect node failures using heartbeat messages. Service Oriented Architecture is another example where a number of services communicate using a complex chain of RPC calls. Since the underlying communication takes place in the form of raw packets, often the program behavior can potentially change if the packets are received out of order or packets are lost in the network. Therefore, in this context, both the incoming data and the send/receive order can potentially impact program behavior and consequently increase the complexity of verification methodology.

An additional challenge is attributed to concurrency. Most modern systems use multi-threading to achieve better performance. In such a case, the outcome of a program is a function of how individual threads are interleaved together. Reasoning about such interleaved execution paths is hard and adds another layer of complexity to the process of verification.

Given the above challenges, we were curious about the usability of existing symbolic execution tools for verification of system software. In particular, we were interested in understanding how easy it is to use symbolic execution to verify systems software that interacts with the networking stack. We, therefore, analyzed Snort using the popular symbolic execution engine named KLEE [3]. We documented the challenges that we encountered along the way. We also looked at recent research papers to understand how they approach the problem of verifying such code.

*These authors contributed equally to this work

.

2. INTRODUCTION

Packet processing functionality in the network like load balancing, network address translation, and intrusion detection and prevention systems have historically been implemented using hardware devices called *middleboxes*. This approach has several drawbacks including lack of flexibility in moving across different vendors, high operating costs and requirement for tedious manual configurations. Hardware upgrades are rare and as a result network operators don't have a lot of flexibility in changing functionality. These problems can be solved by implementing the same functionality in software. Such packet processing units are called *Network Functions* (NF).

NFs are deployed on commodity hardware units. Moreover, the usage of software stack for implementing middlebox functionality adds agility to the development and deployment process. Verification for NFs is particularly important because NFs are on the critical path of the entry of packets into the system. Moreover, given their strategic position in the execution path, NFs can be leveraged to detect DDoS attacks. NFs are latency critical, and therefore the code has to be verified for both functionality and performance characteristics. Since NFs are executed on general purpose processors, performance verification becomes important.

Networks, in general, are hard to debug and reason about, and as a result, they are associated with two distinct categories of verification: Network Verification and NF verification. Reachability analysis between nodes in a given network topology and presence of loops are examples of questions that network verification seeks to answer. Network verification looks at the holistic view of individual network configurations of all the devices connected together. Network Function verification, on the other hand, is concerned about the semantics of individual NFs and its properties like susceptibility to buffer overflows, memory leaks etc. It is independent of the topology and workload characteristics of the network.

In this project, we gauge the usability of existing tools for

NF verification. To this end, we analyze Snort using a state-of-art Symbolic Execution Engine (SEE) called KLEE. Rest of the report is organized as follows: Section 3 talks about relevant background information about Snort and KLEE. Section 4 focuses on the hurdles we encountered when integrating KLEE with complex pieces of software. Section 5 elaborates about out evaluation results for Snort 1.3. We then talk about relevant research papers and elaborate about their strengths and limitations. Lastly, we conclude about our learnings both about the practical aspect of using KLEE and usability of such tools.

3. BACKGROUND

3.1 Snort

Snort is an open source Intrusion Detection System (IDS). Snort was first published in January 1999, and since then has evolved gradually into a more complex system. Later versions of Snort support the below major functionality:

- **Packet Sniffer:** Snort can be configured to listen to a network interface. It then reads the packets and performs actions as specified by the user.
- `pcap` **File Replayer:** We can replay `pcap` files using Snort. This functionality comes in particularly handy for testing NF codebase against different workloads for performance benchmarking and for testing code in general.
- **Intrusion Prevention System (IPS):** Early versions of Snort only created a log/alert messages in response to the incoming packets. Later versions of Snort can drop packets in accordance with the user provided configuration.

Developers can specify what actions Snort is expected to take in response to different packets using Snort's rule language. Snort's rules consist of below major components:

- Action (Example log, alert)
- Protocol (`ip`, `tcp`, `udp` etc.)
- Source IP address and port
- Destination IP address and port
- Direction operator: `->` is unidirectional operator and <> is bidirectional operator. It indicates whether rule is applicable to traffic flowing in a single direction or both directions.

Below is a sample rule:

```
log tcp any any -> 192.168.1.0/24 :6000
```

This rule indicates that all TCP traffic from any source IP address and source port with a destination port less than 6000 should be logged. Similarly, the developer can create alert rules for different types of incoming traffic and source and destination IP address and port combinations.

3.2 KLEE

KLEE [3] is a symbolic execution engine capable of automatically generating tests for the code to be verified. KLEE is limited to analyzing C code.
At a high-level, KLEE has two goals:

- Hit every line of executable code in the program and
- Detect at each dangerous operation (e.g., dereference, assertion) if any input value exists that could cause an error.

KLEE achieves the above goals by executing programs symbolically. Unlike normal execution, where operations produce concrete values from their operands, KLEE generates constraints that describe the set of values possible on a given path. When KLEE detects an error or when a path reaches an exit call, KLEE solves the current path's constraints to produce a test case that will follow the same path when re-run on an unmodified version of the checked program (e.g, compiled with `gcc`).
To demonstrate the usefulness of their tool, the authors ran KLEE on unmodified GNU Coreutils [1] and found 3 unique bugs, which went undetected for 15 years.

4. CHALLENGES

We ran into a number of technical glitches when executing Snort with KLEE. We went through many iterations of trying to get different versions of Snort - starting from the latest version (Snort 3.0) to Snort 1.3, while also trying to get `iptables` to work with KLEE. In this section, we talk about the many issues we ran into, the steps we took to work around them, and the lessons we learned from that exercise.

- **Snort 3.0:** This is the latest version of Snort, and is currently available as a beta release. Snort 3.0 is modular, multi-threaded, and has a lot of features that make it easy to use and deploy.
  We tried to use Snort 3.0 for the term project, but soon realized that KLEE does not support multi-threaded program analysis. We could not proceed any further with this Snort release and tried to find an earlier version to test.
- **Snort 2.9.12:** This is the current stable release of the product. We tried to analyze Snort 2.9 with KLEE, and soon realized that it has a lot of dependencies - all of which need to be recompiled with `WLLVM` (Whole program LLVM) to generate KLEE supported Intermediate Representation(IR).
  We were able to convert required dependencies into LLVM IR to run symbolic execution on Snort. But, we were only able to execute the command to list the version. When we tried to make more changes to Snort to exercise a more meaningful code path, we ran into header file conflicts. We could not get these resolved, because of the inherent complexity of Snort code.
- `iptables` Given that we were not able to run a complex code base with KLEE, we decided to try a simpler firewall program. The most viable option that emerged was iptables, which is a user-space utility to install Netfilter kernel rules.
  We started compiling iptables code and that of its

many dependencies using wllvm. We succeeded in transforming all the dependencies to the LLVM IR. But, when we started running actual rules, we quickly realized that there is limited support for sockets in `uclibc` (standard C library encapsulating important functions like I/O, string utilities, socket calls etc.). The socket functions under the cover use inline assembly code. KLEE does not support symbolic execution of inline assembly code, and as a result, it does not support socket binding.

After running into the above challenges, we moved to simpler versions of Snort. We were able to evaluate Snort1.3 with KLEE and the relevant details are elaborated in the following section.

5. EVALUATION

After running into the above problems, we looked at recent papers for inspiration on how researchers have approached the problem of verifying complex pieces of software. When doing a literature survey, we came across a HotNets 2016 paper [7], where Wu et al have analyzed there proposed methodology against Snort 1.0. Snort 1.0 was released in 1999, whereas the paper was published in 2016. We then realized that we should be focusing on analyzing early releases of Snort. We tried to compile different versions of Snort varying from 1.0 to 1.6.1. Of these releases, only Snort 1.3 compiles with a standard `gcc` compiler. Other releases do not compile with `gcc`. We, therefore, proceeded with analyzing Snort 1.3 with KLEE. All references to Snort in the evaluation section implicitly refer to version 1.3.

Snort code is organized into the following high-level modules:

- **Packet Decoder**: Packet headers for protocols like Internet Protocol (IP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP) etc. have very specific header formats which have to parsed out by Snort. This module encapsulates the details of this information extraction in order to:
  - Process rules specified by the user since rules are protocol specific and
  - Maintain state for how many packets of each protocol have been processed.
- **Rules Engine**: This module takes in the rules file provided by the user as an input. It parses and validates the rules, and creates a forest of rule chains, where each forest comprises of rules applicable to individual protocols.
- `pcap` **File Replayer**: This basically opens file descriptor to a `pcap` file. It reads the input file and parses out the header and payload from the input file.
- **String Utils**: Packet Decoder and rules engine process strings and therefore, common string utilities like tokenizing strings, searching for substrings etc, are abstracted out into a common utility class.
- **Logging Functionality**: This performs two-fold functionality of:
  - creating relevant alerts in accordance with the rules specified by the network operator and
  - logging relevant debugging information for the developers

Snort has one external dependency on `libpcap` library. It is used to open a sniffing session on an Ethernet interface.

We executed KLEE on the snort library in two phases:

1. Phase 1: Testing the end-to-end functionality of Snort:
Snort provides two high-level functionalities namely sniffing incoming packets and replaying `pcap` files. We primarily verified the replay feature. We were not able to operate Snort in sniffer mode due to limited socket support in KLEE. Coverage results for replay feature are shown in Table 1.
For our analysis, we had made the packet symbolic, in order to generate sample packets that will execute different code paths. The instruction coverage includes code lines executed in associated libraries as well. Snort uses only the socket connection code of libpcap and does not invoke rest of libpcap code. When we had first executed Snort with KLEE, we had 10.14% code coverage. We improved our coverage by using a leaner version of libpcap library. We also refactored Snort code and removed dead code to further improve code coverage.

2. Phase 2: Testing individual libraries of KLEE:
In order to bypass limited socket support in KLEE, we took the bottom-up approach to verify KLEE. We verified the foundational components namely the packet decoder, rules engine processor, `pcap` file replayer and string utilities. Logging utility simply redirects strings to log files and therefore does not encapsulate bulk of the processing logic of Snort. We, therefore, chose to focus on other building blocks of Snort.

- **Packet decoder and `pcap` File Replayer:** We verified these modules together because of their intertwined functionality. For these modules, we made the packet which is a character array symbolic and verified decoding of IP packets. We took two different approaches to this due to the observation that packet headers have a specific format. We were curious if giving KLEE hints about payload length, protocol etc. will cause a change in coverage results. Our observation is that instruction coverage improves marginally whereas branch coverage is the same in both cases. These results are shown in Table 2 and Table 3.
- **Rules Engine:** For verifying the rules engine, we made individual rule strings symbolic, with the expectation that KLEE will generate sample rules that will exercise specific code paths. Although KLEE generated more than 150 thousand test cases, it was able to achieve code coverage of only 16.87%. This is shown in Table 4.
  We tested the replay functionality with a test rules file and measured the resultant code coverage. We were able to cover 52.98% of the rules engine processor code with only 25 rules. This is not surprising since rules can be free-form strings, and developers or network engineers can inherently do a better job of crafting such

KLEE: done: total instructions = 1921059
KLEE: done: completed paths = 2203
KLEE: done: generated tests = 2203

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
1921059	26.18	29.43	22.75	24333	93.73

Table 1: Coverage results for Snort when operated in pcap replay mode

KLEE: done: total instructions = 23177300
KLEE: done: completed paths = 12921
KLEE: done: generated tests = 12921

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
23177300	66.97	37.46	26.55	6505	15.60

Table 2: Coverage of Packet decoder with packet length and header constraints

free-form strings to test different code paths.

- **String utilities:** There are three helper functions used by Snort:
  - **Substring search functionality:** This is used to find a pattern in a given string, where the search is not based on a regular expression. For the purpose of our analysis, we made both the pattern and the string to be searched symbolic. The results are in Table 5
  - **Pattern match functionality:** This is simply a different implementation of the substring utility mentioned above. The results for KLEE instruction coverage are shown in Table 6.
    When we looked through the larger code base, we realized that this is dead code and we removed it. Verifying code from bottom-up gave us the chance to analyze the code more closely. We eliminated dead code and also refactored code to minimize duplication of code. It also increased our confidence in our verification results.
  - **String Tokenizer:** This is used to tokenize the input string using a specified separator. Both the string to be tokenized and the separator were made symbolic for our analysis. The results from this are shown in Table 7

Thus, by using a mix of bottom up and end-to-end functionality verification, we analyzed different parts of Snort using KLEE.

KLEE: done: total instructions = 15693845
KLEE: done: completed paths = 8564
KLEE: done: generated tests = 8564

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
15693845	51.25	37.35	26.55	6493	24.51

Table 3: Coverage of packet decoder with no header and protocol constraints

KLEE: done: total instructions = 7316565
KLEE: done: completed paths = 156764
KLEE: done: generated tests = 156681

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
7316565	31.78	16.87	12.39	13328	39.05

Table 4: Coverage results for Snort's Rule Engine

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
275412	13.77	43.41	21.43	463	90.13

Table 5: Coverage for substring utility in String Utils

6. RELATED WORK

At the point where a number of our initial results to execute KLEE to analyze software end-to-end failed, we looked at recently published papers to gain insight into additional applications cases for KLEE. Below is a brief review of interesting papers:

6.1 Automatic Synthesis of NF Models by Program Analysis

This work [7] was done at HP Labs, and describes a new tool called NFactor that refactors and slices code in order to generate its forwarding model.

This paper identifies that running symbolic execution on an unmodified NF is not easy, and therefore, it is beneficial to come up with a model of the NF first. Developing accurate models of arbitrary NFs is challenging for a number of reasons including usage of non-standard and proprietary NFs. They develop a technique to slice the programs and analyze state tainting to come up with accurate models of the NF execution.
While this work is only tangentially related to the work we did for the course project, it caught our attention because they use their NFactor tool to create a model for Snort 1.0. This is the only paper that runs Symbolic Execution on a commercially used NF, but even here the Symbolic Execution is run on a reduced slice of the original codebase.

NFactor makes two standard non-limiting assumptions on the code structure:

- Since the NF program needs to continuously process incoming packets, there exists a packet processing loop. The same assumption has been made in StateAlyzer [4].
- NF programs usually use standard library or system functions to exchange packets with the OS kernel/network devices - thus, NFactor leverages this knowledge to locate packet read/write statements in the program. NFactor identifies the variable that stores a packet by

KLEE: done: total instructions = 368887
KLEE: done: completed paths = 7570
KLEE: done: generated tests = 7570

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
368887	31.34	41.04	17.57	385	91.14

Table 6: Coverage for pattern matching in String Utils

KLEE: done: total instructions = 7041845
KLEE: done: completed paths = 7738
KLEE: done: generated tests = 7711

Instrs	Time (s)	ICov (%)	BCov (%)	ICount	TSolver (%)
7041845	21.03	40.25	26.99	5230	40.22

Table 7: Coverage for string tokenizer in String Utils

fetching the return value of the packet input function or the argument of the packet output function.

Using this slicing method, they were able to get a code coverage of about 4%. This is significantly lower than the code coverage we got - 29%.

The most curious thing about this work is not the clever program analysis tools to identify packet and state slices, but the choice of the Snort version they analyzed. The version of Snort used for this work was 1.0 (released in 1999), and there is no clear rationale provided for why they used this version. It might be the case, as with most research, that the engineering hassles of making these big software projects work are not reliably conveyed in the publication.

6.2 Formally Verified NAT

Network Address Translator (NAT) modifies the incoming packets by rewriting IP addresses and the associated checksums in the packet header. Verifying NAT is hard because it is a stateful NF. It maintains a flow table and has to expire entries in the flow table after a configured timeout.

In order to formally verify NAT, the authors used two different formal verification methods - theorem proving and symbolic execution. They first rewrote NAT by abstracting away the stateful code and the associated data-structures into a separate library called `libVig`. This separation of concerns sets the base for applying different verification methods.
To verify the `libVig` library, its code is first annotated with pre and post conditions. The specified conditions are then fed to Verifast which does the actual task of verification. According to the paper, it took researchers two-person months to verify the code. After the stateful code was verified, the remainder of the stateless code was verified using KLEE. They used the premise that a system composed of individual formally verified components is automatically verified.

Apart from a verified NAT, [8] makes two contributions to the verification and NF research. The first is the concept of lazy proofs to bind together individually verified components. The second is the `libVig` library itself, which has the potential of being reused across NFs. However, from a usability perspective, it is hard to ignore that the authors of this paper had to rewrite the NAT code from the ground up. Moreover, the `libVig` library does not use concurrent data structures. As a result, the verified NAT does not work with multiple threads.

6.3 Formally Verified NAT Stack

The research group that worked on [8] also published [6] wherein they extended the toolchain to verify the kernel-bypass framework and a NIC driver in the context of a NAT. Zaostrovnykh et al. presented a formally verified NAT where they used a verified library to implement the NAT. But, they assumed that the underlying layers - the kernel bypass mechanism provided by Intel's Data Plane Development Kit (DPDK) [2] and the NIC driver - work correctly. While this is a common practice, it is not necessarily true.

In [6], Pirelli et. al. model the C library and the underlying hardware, by which they were able to remove the NIC driver and DPDK from the Trusted Compute Base. They follow a pragmatic way in deciding which parts of the stack to verify, and claim that code that is immature (and thus likely to have bugs) and code that has a high potential for reuse in future NFs should be the prime candidates for verification.

They also note that while kernel-bypass frameworks such as DPDK may be large, writing a basic NF requires only a small subset of DPDK: initialization, receiving packets, and sending packets. This subset contains a lot of code, but it has simple control flow and almost all is simple operations such as reading or writing to device registers.

Instead of verifying DPDK as is, they use a lot of clever changes to make it more amenable to symbolic execution; Ring buffers are replaced with one item structures; inline assembly is avoided in the code; single instruction, multiple data (SIMD) instructions support was added to the KLEE execution engine. Also, since KLEE cannot deal with multithreading, they implement their NAT as a single threaded application and do not use DPDK's thread related features.

While this paper is a good attempt to verify all the components of the stack, it is important to note that the researchers have not verified 100% of the code base. They reduced the scope of code to be verified and then used symbolic execution methodology to verify rest of the code.

6.4 Automated Synthesis of Adversarial Workloads for Network Functions

This paper [5] addresses the problem of performance verification. NF functionality was traditionally implemented using specialized hardware, but the move to commodity hardware has introduced the additional problem of performance unpredictability. NF performance verification is critical because most NFs are latency sensitive, and since NFs are the entry point of any system/service, they can impact the SLA of the entire system.

The main contribution from this paper is a system called CASTAN (Cycle Approximating Symbolic Timing Analysis for Network Functions) that uses KLEE to generate `pcap` files to stress the system. CASTAN addresses the problem of path explosion in symbolic execution by using two search heuristics: maximizing number of instructions executed and maximizing cache misses. CASTAN modified KLEE to use the above search heuristics. The resultant test cases were then used to generate `pcap` files that stress the system, which can then be used by developers to identify performance bottlenecks in the system.

CASTAN is novel in its use of formal verification methods to verify the performance of a system. To evaluate CASTAN, the authors measure throughput and latency degradation under different workloads including the CASTAN generated workload. However, their evaluation does not produce strong results - Unirand workload outperforms CASTAN for

all NFs that were used to evaluate CASTAN. Unirand workload is generated by simply using a random number generator for source and destination IP addresses and ports. While CASTAN is an interesting application of symbolic execution, the key take-away from the paper is that the most complicated solution isn't necessarily the best one. In this case, one must evaluate the cost benefit of using symbolic execution for it's not universally more likely to lead to a better solution.

7. DISCUSSION

We encountered a number of problems when verifying Snort using KLEE. We also looked at recent research papers and analyzed their strengths and weakness. We summarize our key takeaways below:

- *Lack of support for multi-threaded software*: KLEE has the inherent limitation that it only works with single-threaded code. This limitation gets propagated to all systems that rely on KLEE, as is evidenced in the many systems discussed in Section 6.

- *Lack of support for sockets*: KLEE does not support inline assembly code and therefore, KLEE crashes when trying to bind to a socket. As a result, it is hard to verify code without modifying important networking code.

- *The need to modify code specially for verification*: Often it is required to rewrite the code-base in order to verify a NF, as seen in [8]. This is tenable for smaller pieces of software, but with larger components and legacy applications, it is a non-trivial exercise to re-implement the applications from scratch.

- *Symbolic Execution is most useful when used from initial stages of development*: As a corollary from the above arguments, it is clear that it is easier to verify simpler pieces of code. Therefore, it is beneficial for the developer to use such tools from the early stages of development. Incremental verification of smaller pieces of code and reorganization is easier when done in an agile manner, compared to the case where developers have to invest lots of effort to verify a large and complicated code-base that's at an advanced stage of development.

- During the initial stage of the project, we were rather surprised that research papers [7] published in HotNets 2016 used an ancient version of Snort (Version 1.0, Released in 1999) for analysis. They do not explicitly explain the rationale for their choice of version, but it certainly stands out as a red flag.
  Our conjecture is that they were either limited by a) the code complexity of later versions of Snort code or b) Some feature of later versions of Snort is not compatible with KLEE. Regardless of the underlying reason, it is hard to ignore that even recent papers do not use the latest versions of open source NFs for their analysis. Better verification tool support and community best-practice guidelines for porting large code-bases to a form that's amenable to symbolic execution are sorely needed.

8. CONCLUSION

We learned about the details of how symbolic execution engines work, and about the recent research efforts in verification of the software data plane. While we stumbled into a number of technical problems while verifying NFs, each roadblock gave us deep insights into the usability aspects of KLEE. We were able to partly verify Snort's functionality, particularly its ability to replay `pcap` files. We also studied all the sub-components of Snort and integrated them with KLEE, to get a better understanding of Snort, and how to structure code in a way that's amenable to verification.

9. REFERENCES

[1] Coreutils - gnu core utilities. `https://www.gnu.org/software/coreutils/`.

[2] DPDK: Data plane development kit. `https://www.dpdk.org/`.

[3] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[4] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In *NSDI*, pages 239–253, 2016.

[5] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 372–385. ACM, 2018.

[6] S. Pirelli, A. Zaostrovnykh, and G. Candea. A formally verified nat stack. In *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks*, pages 8–14. ACM, 2018.

[7] W. Wu, Y. Zhang, and S. Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 29–35. ACM, 2016.

[8] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 141–154. ACM, 2017.