Ecole polytechnique fédérale de Lausanne

Real-world verified network firewall

supervised by Solal Pirelli

Semester Project

Patrice Kast
Elvric Trombert

June 15, 2022

Contents

1 Introduction

In this semester project we aimed to extend the potential use of the tool "Klint" from the paper "Automated Verification of Network Function Binaries"[1] presented at NSDI'22. This tool can be used to formally verify network function implementations (NFs) by just using a compiled binary file together with a specification file.

In this report we will give a brief user guide on how the tool Klint can be used. We will then move on to the work and discoveries we have made through this project, to finally talk about the improvements we foresee for the tool in the future.

2 User guide

Klint can be broken down into two sections, one is the library that must be used when coding the NF and the second part is the library that must be used when writing the specifications.

2.1 Understanding how to code a network function for Klint

2.1.1 Library Documentation

Originally it was hard for us to make sense of the capabilities of the library as there was no formal documentation on how to use it. Our approach consisted mostly on looking at previous NF examples and reverse engineering them to understand how they are coded.

To assist future users we have documented the existing library using the doxygen code documentation guideline [2]. From there, the documentation for the Klint library can be generated.

2.2 C library to use

2.2.1 nf_init(device_t device)

This is the initiation function that must be declared in every NF. This is where all the memory allocations must occur. As we cannot allow our NF to fail due to a lack of memory space, the space cannot be dynamically allocated but is always statically allocated on initialisation.

2.2.2 nf_handle(struct net_packet packet)

This function is invoked when our NF receives a packet. The whole logic of the processing should happen in here.

2.3 Data structures

The most important thing to understand in this section is the concept of memory ownership [3]. When we started the project the memory ownership would be partially transferred to the data structure when used and it was up to the developer

to allocate the necessary space required for each data structure. This hindered the development process as it made it more complex.

To address that we declared new data structures that take care of allocating the necessary memory space they will use. Furthermore data passed to them will be copied to memory areas controlled by the data structure avoiding memory ownership transfers.

2.3.1 os_config_get

To prevent having to hard code any information directly in the implementation of a NF, a config file, located in the same folder as the binary, can be filled with fixed-size values used for the processing of packets.

To read them into a variable during run time, a series of functions is provided by the library.

The config file format is as follows:

```
{ "[name of value]", [value (must be a number)] },
{ "[name of value]", [value (must be a number)] },
```

2.3.2 lpm (longest prefix match)

The longest prefix matching (short lpm) is a data structure needed to be created for NFs like firewalls, where IP filtering rules need to be created. This is done by most IP address rules in firewalls, such as iptables [4] in the Linux kernel. The *lpm* behaves like a regular map except that when entering a key value pair, we must also enter a key mask. That key mask will define the number of most significant bits that will be used to compare the key with other keys. When searching for an item in the map, the key we entered is compared with the most significant bit of the keys present in the lpm, and the entry corresponding to the match key with the longest common prefix with the inputted key will be returned.

2.3.3 map

This is a data structure that works like a regular map. It takes as input a pointer to a memory area of a predefined size in the initiation stage of the nf (2.2.1) for the map value. We figured out during the development process, that these values cannot yet be used in formal verification, since the tool cannot handle nested-maps, which could for example be used like bitmaps.

2.3.4 index pool

The index pool is a handy data structure which can be used to retrieve available indexes in a pre-allocated array with a built in expiry counter to free up old and unused objects.

2.3.5 memory allocation

All memory allocation must be done via the `os_memory_alloc` function as Klint uses it to be able to track the allocated memory. Other functions such as `malloc`, `calloc` must not be used.

2.4 Specification library

In order to write specifications of the binaries, there exists a helpful python3 library, that makes working with maps and reasoning about them more easily.

2.4.1 Defining Datatypes

The first step in writing a specification file is to define the map and the datatype which is used as its key and value. This can be easily done, by defining a struct with the corresponding bit-size information of its attribute.

```
MapKey = {
  'width': 'size_t',
  'is_dest_ip': 8,
  'dst_port': 16,
  'ipv4_addr': 32,
}
```

Figure 1: KeyMap datatype declaration for the iptables.

2.4.2 The spec() function

Afterward defining the data structure, the map can be instantiated in the main function `def spec()` as it can seen in the next code snippet:

```
def spec(packet, config, transmitted_packet):
    rules = Map(MapKey,...)
    assert(True)
```

The passed parameters are explained below:

1. `packet`: this is the original packet, which gets passed to the NF and triggers the processing.
2. `config`: the config array can be used to directly access fields of the static config file.
3. `transmitted_packet`: This is the packet, which is being sent out of the NF after processing.

2.4.3 Writing the assertions

If we now want to check for example, that no packet was sent out when an command packet on the command interface was received, we can do it as shown below:

```
if packet.device == config['command device']:
    assert transmitted_packet is None, "Command packets should not be forwarded"
```

To now assert a internal state change of the map (called rules in our example), we can now create a new struct and search for it in the map:

```
command = get_header(packet, Command)
rule = {
    'width': command.mask_width,
    'is_dest_ip': command.is_dest_ip,
    'dst_port': command.dst_port,
    'ipv4_addr': command.ipv4_addr,
}

if rules.old.length < config['capacity']:
    assert rule in rules, "The rule should be in the map if it's valid and the
    ↪  map has space"
```

If we now receive a packet on the listening interface and we want to assert, that it is only forwarded if a specific rule exists in the map, we can do it like below:

```
if packet.device == config['check device']:
rule = {
    'is_dest_ip': constant(0,8), # Source
    'dst_port': packet.tcpudp.src,
    'ipv4_addr': packet.ipv4.src,
}
if rules.forall(lambda k,v: (k.is_dest_ip != rule["is_dest_ip"]) | (k.dst_port !=
↪  rule["dst_port"])):
    assert transmitted_packet is None, "not routed packets should not be
    ↪  forwarded"
else:
    assert transmitted_packet is not None, "routed packets should be forwarded"
```

2.4.4 Hint & Tricks

In our example we made no use of the values stored in the map. Some other hints for writing specifications are listed below:

1. If an element, which is being searched for, is created as a struct, attributes needs to be accessed in the array-bracket notation instead of properties.
2. If attributes of a key in the map need to be compared, always use properties and not the array-bracket notation.
3. Structs can not (yet) be shifted - use concats instead for this purpose.
4. It is important to always check for the negation of a stement in the `forall` loops.
5. If the Ternary conditional operator needs to be used, use the `if_then_else` block to model these conditions in order to help Klint understand.

3 Work and discoveries

3.1 Using rust

At the start of this semester project, there was a rust implementation of network functions present in the git repository. Since it was more like c-code-to-rust translations with a lot of unsafe statements, we decided to try and implement a network function fully written in the memory safe rust code.

For this, we started rewriting the bridge in rust.

The first barrier to the use of rust was the presence of library specific functions, such as map, lpm, custom memory allocations, that are coded in C and must be used for Klint to be able to make sense of the NF built. To address this problem, we used `rust-bindgen`[5] in order to convert C header files to rust files acting as the bridge between the two environments.

The next problem arose when starting to code the NF using the bridged rust files. We realized that the memory management of rust could not keep track of the memory that was being used when calling the C libraries of Klint and vice versa. Hence the feature that triggered our original interest in rust, namely its memory safe property was not applicable for the project, so we decided to move back to C.

3.2 Looking at Iptables

As mentioned previously, iptables is the basic default firewall present in the Linux kernel. Hence we decided to take some of its functionalities and implement them step by step as individual NFs that can be formally verified by Klint. Namely an IP address blacklist filter, that based on the source and destination of packets would decide if a given packet was going to be let through or dropped. This got eventually extended to IP address whitelist filtering and finally in a last iteration to an IP address inclusive port whitelist filtering.

3.2.1 Obstacle of data structure

As we have mentioned earlier the main method used today for IP address filtering is the use of a longest prefix match data structure. Yet at the time we started this project, such data structure was not yet available. To mitigate this problem we switched to the map data structure for the time being.

Using a map on its own was not a problem. The issue arose when Klint missed the assertion link between checking if an element is in the map and then removing that said element from the map. Further more the map available was only able to hold primitive value types which limited its use. Lastly the map created was not responsible for storing the data. Hence we had to define and allocate all the space to be used by the map manually and deal with partial memory ownership transfers to the map.

3.2.2 Building new data structure

To avoid the problems mentioned above we decided to write new data structure contracts and definitions that are likely to be used for more advanced NF. This is where a new map structure was created, one that handled the memory allocation as well as an LPM data structure.

3.2.3 Designing the IP address blacklist commands

One big challenge of formal verification is that loops can lead to path explosion. Thus we could not make use of the `config` file to hold the list of IP addresses as their number can vary. This is where we created a new type of packet structure, a packet that has as payload, a command that can be interpreted by the NF. This solution worked as Klint is already familiar with packet structures, so we just had to change the way the NF would handle different types of packets.

We interpreted packets coming from a predefined command-interface as command packets and used these to add or remove rules on the fly during run time.

3.2.4 Writing the specifications

This was the toughest part of the project. In order to use the LPM in our NF we had to write its specification abstracted as a regular map in the Klint verification tool. This made it quite challenging when writing the specifications as we had to be aware of the method used to abstract the LPM as map in order to be able to check if a given element was or was not in the LPM.

Another challenge came from verifying if a given key was present or not in the LPM via map abstractions. We had to make use of the `forall` method and create new functions such as the `if_then_else` within it in order for Klint to be able to follow the execution path.

3.3 Building more advanced NF

As we worked with iptables, it came to our attention that in order to predict what kind of improvements could be interesting to look at in the future. A good way is to design more NFs and analyse what functionality currently missing might be required.

3.3.1 IPv4 DHCP Server

One of these NFs is a DHCP server. It was built in order to provide the basic IP address allocation in a network following the RFC 2131 [6] convention.

What we discovered when writing this NF is the potential need to provide a data initialization function as this NF requires available IP addresses to be generated. Work-around would still exist but for more advanced NFs, it is likely that the ability to initialize values in the allocated memory would eventually be needed.

3.3.2 DNS Server

Another NF that we implemented was a simple DNS server, capable of replying to A-record requests. It parses an incoming DNS request packet and reads out the required information. The associated data structures of this process are closely modelled as defined in the [RFC 1035][1] except the DNSSEC flag which got extended in [RFC 2065][2].

It parses the question section and generates a resource record by reusing the memory space of the incoming packet, in order to avoid allocating any new memory during run time.

By having a command interface, new DNS records can be added ore removed on-the-fly by sending packets which follows the structure of a `net_dns_record`.

As an early prototype, a static configurable DNS server was attempted. The problem, which aroused there, was the fact that it ended up being difficult to parse a non-fixed number of string records defined in the config file.

4 Potential Future Work

4.1 Debugging during formal verification steps

One of the things we struggle the most with, besides writing the specification for our NFs in order to formally verify them with Klint, was to understand the error that appeared when the formal verification failed, in order to figure out what changes are needed in our NF's code. Our greatest limitation though when using the tool was to get more familiar with the errors coming from Klint. One feature for Klint, that we think could be very useful, would be to provide a concrete input example that cause the verification error in addition to the error message. This could then be used to check what sort of input or state would lead to this specification violation.

4.2 Documentation on how to write Klint specifications

As mentioned above, one challenge with Klint is learning to write the specification and understanding the usage of the tool in general. One way to improve that would be to do something similar to what we did for the C library used with Klint in §2.1.1. Allowing the developers to get further insights on what functions and data type to be used when writing the specifications.

4.3 Extend support to more data structures

We realised in this semester project that for larger NF more sophisticated data structures need to be made available so that formal verification using Klint could be applied to more advanced NF e.g linked list, stacks.

[1] https://datatracker.ietf.org/doc/html/rfc1035
[2] https://datatracker.ietf.org/doc/html/rfc2065

4.4 Add special default function to manage NF internal logic

Another interesting thing that would be a great feature when coding more advanced NF is the need to manage its internal state. In order to do so we created a specific type of packet responsible for carrying commands to dynamically change the state of our NF. One proposition that was made by our supervisor was to add a new function to all the NF built with Klint that will be responsible for handling these change of state.

As for now, the developer is responsible for handling this logic directly in the `nf_handle` function, making the function perform two separate tasks:

1. manage internal state change of the NF
2. handle regular network packets.

4.5 Add support for persistent storage

Following that same logic, one of the challenges will be to manage the internal state of these NFs. At present when the NF is launched, it is up to the user to send it a specific list of commands in order to change its internal state. A state that is internal to the NF and cannot be checked against our specification. The possibility of persistent storage of state would allow a static state to be defined prior to the launch of the NF, allowing it to be included in the specification used to formally verify the tool with Klint.

4.6 Improve packet handling flexibility

When creating the DHCP and the DNS server, we realised that we would also need to craft and send out new packets of various length containing information for the client in order to follow the official RFC. Thus allowing sent packets to have a different size than the one received may eventually be required.

5 Conclusion

The idea of formally-verifying a binary will definitely be an essential part of secure & reliable firewalls in the future. This project enabled us to get an overview of this topic. Even though we the learning curve was steep at the beginning. We now have a greater understand of what is happening under the hood and = why something can or cannot be formally verified.

Overall five NFs were created and 3 formally verified. The formally verified include: IPv4 address filtering black list, IPv4 address filtering whitelist and Port plus IPv4 address filtering whitelist. The two that were coded but not yet formally verified are: DHCP server and DNS server.

Still, the topic is big and we are sure that we only scratched the surface of what would be possible in the field of automated formal verification.

References

[1] Solal Pirelli et al. “Automated Verification of Network Function Binaries”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 585–600. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/pirelli.

[2] *Documenting the code*. https://www.doxygen.nl/manual/docblocks.html.

[3] Elaf Alhazmi, Abdulwahab Aljubairy, and Ahoud Alhazmi. “Memory Management via Ownership Concept Rust and Swift: Experimental Study”. In: *International Journal of Computer Applications* 183.22 (Aug. 2021), pp. 1–10. ISSN: 0975-8887. DOI: 10.5120/ijca2021921572. URL: http://www.ijcaonline.org/archives/volume183/number22/32054-2021921572.

[4] *iptables(8) - Linux man page*. https://linux.die.net/man/8/iptables.

[5] *bindgen*. https://github.com/rust-lang/rust-bindgen.

[6] R. Droms. *Dynamic Host Configuration Protocol*. RFC 2131. Bucknell University, Mar. 1997, pp. 1–45. URL: https://datatracker.ietf.org/doc/html/rfc2131.