

SCIENTIFIC STUDENT CONFERENCE

Károly Kecskeméti

Simplifying the development of in-network security applications with P4RRROT

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF INFORMATION SYSTEMS



Author:

Károly Kecskeméti

Computer Science MSc

2nd year

Supervisors:

Sándor Laki, PhD

Assistant professor

Csaba Györgyi

PhD Student

Budapest, 2022

Contents

1	Introduction	3
2	Motivation and background	6
2.1	Code generator tools	6
2.2	P4RRROT	7
2.3	The security landscape	8
2.4	Necessity of quickly deployed security applications	9
2.5	Network security applications	9
2.5.1	Packet filtering(P4Guard)	11
2.5.2	OS fingerprinting (P40f)	11
2.5.3	Port Knocking (P4Knocking)	12
3	Method	14
3.1	Extending P4RRROT	14
3.2	CheckControlPlaneSet command	15
3.3	ReadFromControlPlaneSet command	15
3.4	GeneralBloomFilter data structure	18
3.5	Register commands	20
3.6	Comparison table commands	21
3.7	Digest command	23
3.8	Smaller commands	24
4	Recreating in-network security applications with the use of reusable commands and templates	26
4.1	Hotfixes	26
4.1.1	NTP vulnerabilities	26
4.1.2	NTPGuard	27
4.1.3	Tests and results	27

4.2	P40f	27
4.2.1	Parsing complex structures	28
4.2.2	Tests and results	28
4.3	P4Knocking	28
4.3.1	A more complex version	29
4.3.2	Tests and results	30
4.4	Single touch state machines	31
5	Evaluation	32
5.1	Functional correctness	32
5.2	Evaluating the code generation	32
5.3	Performance and resource usage	33
6	Summary	35
6.1	General conclusions	35
6.2	Future work	35
6.3	Acknowledgments	36
	Bibliography	37
	List of Figures	39
	List of Tables	40

Chapter 1

Introduction

Nowadays cybersecurity becomes a more and more important part of our lives both on a personal level and how we conduct our work. We elaborate on its importance in section 2.3 and in it, we cite a research work stating that a data breach can cost a big company up to 4.35 million USD in 2022. Security professionals are always in a very close footrace against malicious actors and while competing in this race they have to think about the end-users experience as well, since if a service's only goal was to be secure, then we could just not create a service at all.

Thinking about walking the fine line between security and usability can give us a feel about how in need of fast and reliable solutions we are. We believe that in-network solutions offer just that. With the utilization of programmable, high performance network devices we are to able to create applications that filter, manipulate or log the network traffic as it flows through our system, in many cases much faster than we would be able to on a traditional server or other end-point. Since in-network computing is a paradigm that has just started to gain popularity, it's naturally a bit time-consuming to learn all the tricks and create code that executes our dreamt up algorithms. This is why in this work, we take a look at how we can create in-network security applications and contribute extensions underpinning the often used patterns to P4RRROT a P4 code generator tool.

In this chapter we introduce the definitions and logical connections that we feel are integral to the understanding of the topic of this thesis. Our goal is to explore the frequently used mechanisms in in-network security applications and extend our already existing open-source project P4RRROT, so that it can simplify the development of these applications.

First, we introduce Software Defined Networking (SDN) and how modern networks

can benefit from it. Then we introduce some of the technical details, the P4 domain-specific language used to program network devices and how the various targets can present programmers with diverse benefits and constraints. Then, we introduce the already existing code generator tools and our own open-source solution P4RROR. Finally, we briefly present the current in-network security landscape, describe the direction of mainstream research and try to highlight points of possible improvement.

Software Defined Networking is an approach to building networks which has been around for some time but has started to really pick up steam not so long ago. According to [1] SDN's main idea is to decouple the abstraction of establishing network policy from the actual process of applying it. The former is done by the Control Plane while the latter is done by the Data Plane. A similarly integral part of SDN is to provide open interfaces with which it becomes possible to dynamically define and change network behaviour.

Programmable network devices provide the opportunity to create highly use-case specific network applications and to quickly implement desired tweaks in the implemented algorithms. This opportunities' significance becomes really when we factor in the fact that in the case of closed devices the implementation of any new protocol or service needs the backing of the community and most likely will take years. This follows from the fact, that it has to be done by the manufacturer and may include the modification of the underlying hardware.

P4 is an open-source domain specific programming language for the specification of data plane behaviour in network devices. Despite it using a C-style syntax P4 is still a highly domain specific language and as such it uses abstractions and structures not widely known outside the in-network computing community.

In the networking world we can have really fine grained logic and complex algorithms but in the end, at least if we look at the network device as a black box, everything comes down to the interpretation, modification, forwarding or dropping of packets. Based on this, it's not surprising that P4 a language designed to program network devices, has a construction to interpret packets, called parser.

A parser is a traditional state machine with a single start state and two final states. Its purpose is to take the packet as a sequence of bits and extract some or all the bits into headers (basically labeled sequences of bits) while possibly extracting some additional information (metadata). The two final states are called accept and reject and indicate whether the interpretation (parsing) of the packet was successful or not. With the exception of the start and final states (which are logically outside the parser according to

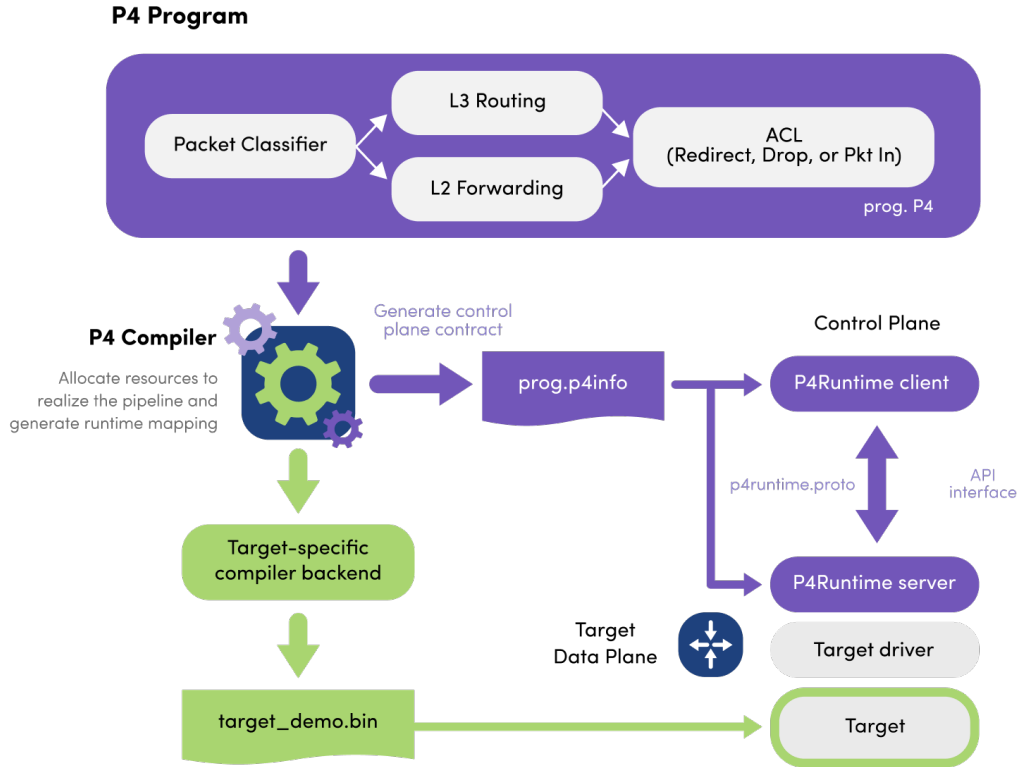


Figure 1.1: Example of the P4 workflow [2]

the specification of P4) all states and transitions are defined by the programmer.

In P4 programs the logic that manipulates the data extracted from the packet is mostly located in so called control blocks. The body of these blocks is said to be very similar to traditional imperative programs by the language specification, an analogy that we find to be very fitting.

P4 is said to be a target independent language, but this does not mean that the same code can run on all devices. The different targets possess a very large scale of strengths and abilities, but naturally these always come with restrictions as well. (In this work we implement in-network applications for the BMv2 software switch and the Tofino Native Architecture.) The great thing about this level of diversity in our opinion is that in most cases it's possible to find just the right device for a situation. The downside is that it can be a really long and difficult process to implement the desired algorithms, hence every tool that promises to make this process faster or easier is a welcome one.

Chapter 2

Motivation and background

In this chapter we take a quick look at the world of code generator tools and then introduce our open-source solution for P4 code generation, P4RROT. After this, we briefly describe the current security landscape and why we think the ability to quickly deploy security applications into a network is of great importance. At the end of the chapter we introduce three security application categories and within each, an application that has been implemented to run on network devices.

2.1 Code generator tools

Numerous works leverage P4 by generating P4 source code to speed up the data plane development or to simplify the description of specific use case-dependent behaviour.

Graph-To-P4 [3] generates boilerplate code for parser graphs. CAMUS [4], FastReact [5] and NetReact[6] perform custom filtering and routing based on application layer data. Lucid[7] is a programming language compiling to Intel Tofino compatible P4 code. It provides event-driven abstractions and a domain-specific type system. V. Nathan et al.[8] propose a network monitoring method involving the compilation of Marple[9] queries to P4 programs.

Sonata[10] offers a declarative interface to express queries for a wide range of common telemetry tasks partially running on programmable switches. It also features method chaining and security use cases.

Poise[11] compiles BYOD (Bring Your Own Device) policies to P4 switch data plane programs and device configuration files to enforce access control based on dynamic run-time context.

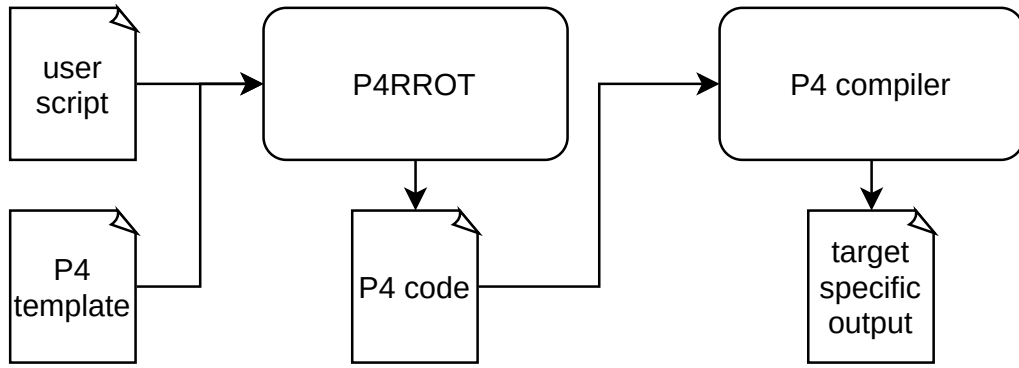


Figure 2.1: The high level overview of P4RROT's usage.

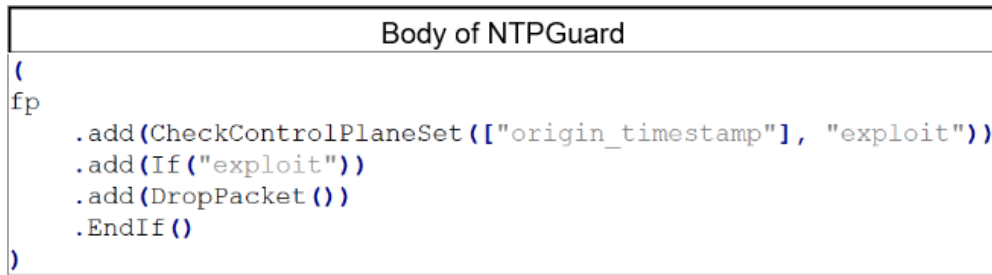


Figure 2.2: Example P4RROT script

In contrast to these problem specific code generators, P4RROT aims to be a general purpose P4 code generator that can be used to reduce the implementation complexity of any in-network computing tasks including security solutions examined in this paper.

2.2 P4RROT

P4RROT is a general purpose P4 code generator for the application layer. P4RROT is implemented in Python which fits greatly into the philosophy of this tool which is to be easy to use and restructure, easily readable but still provide quality. P4RROT is meant to be a collection of structures and high-level commands that the developers can reuse instead of reinventing all the P4 tricks again and again.

An overview of P4RROT's usage can be seen on figure 2.1, an example of a user script's body is depicted on figure 2.2 and the P4 code generated by the CheckControlPlaneSet command can be seen on figure 2.3.

P4RROT can be used in a number of ways. In the day to day work of an experienced P4 programmer it's most likely to be used to take projects off the ground. We mean this in the sense, that it can be used to create the logical core of a network application (which always consumes a lot of time, when done directly in P4) which then can be extended

Example output of CheckControlPlaneSet
<pre>action setter_action_uid4_true() { exploit = 1; } action setter_action_uid4_false() { exploit = 0; } table exact_match_table_uid3 { key = { hdr.genhdr_uid1.origin_timestamp: exact; } actions = { setter_action_uid4_true; setter_action_uid4_false; } size = 1; const default_action = setter_action_uid4_false; }</pre>

Figure 2.3: Example P4 code

manually where that makes more sense. This is because of the fact that P4RROT doesn't aim to be a replacement for P4 programming, but a smart tool that can help lay out the bulk of applications.

For a beginner P4 programmer, P4RROT offers a wealth of learning opportunities. The repository contains a lot of example applications for some of the most popular targets, the implemented commands themselves demonstrate a lot of high level concepts in the language and in many cases it helps the trial and error process of learning, with useful errors.

2.3 The security landscape

Year by year, the importance of cybersecurity becomes more apparent. According to [12] a data breach cost up to 3.86 million USD on average to a big company in 2020, 4.24 million USD in 2021 and 4.35 million USD in 2022.

The average time to identify and contain a data breach was 277 days in 2022 [12]. We believe that just from these numbers it can be concluded adopting a preventive mindset and investing in security is in the best interest of companies and their users alike.

Strong security comes from all across an organization. From the security awareness

of the employees to the technical details of the used devices. About half-way on this scale is how fast the security personnel can tweak the deployed security solutions, whether it's a complicated process to apply a new policy or not. The easier it is to do, the quicker the response to threats will be. And fortunately SDN made quick changes in how we structure and manage our network easier to implement. In this work we aim to pave the road for even quicker changes.

2.4 Necessity of quickly deployed security applications

Zero-day vulnerabilities are vulnerabilities the vendor of a software (or device) is unaware of or just became aware of. Following from this, zero-day exploits are methods an attacker can use to exploit a zero-day vulnerability.

From a security standpoint, a serious zero-day vulnerability found in a broadly used software is almost always an event with great negative impact, since in most cases it takes some time to formulate a patch or workaround solution, hence vulnerable servers and services might have to be shut down for a while, which can mean a loss of money and customer trust.

According to [13] and as you can see on 2.4 2021 has seen an all time high of significant zero-day vulnerabilities exploited by adversaries.

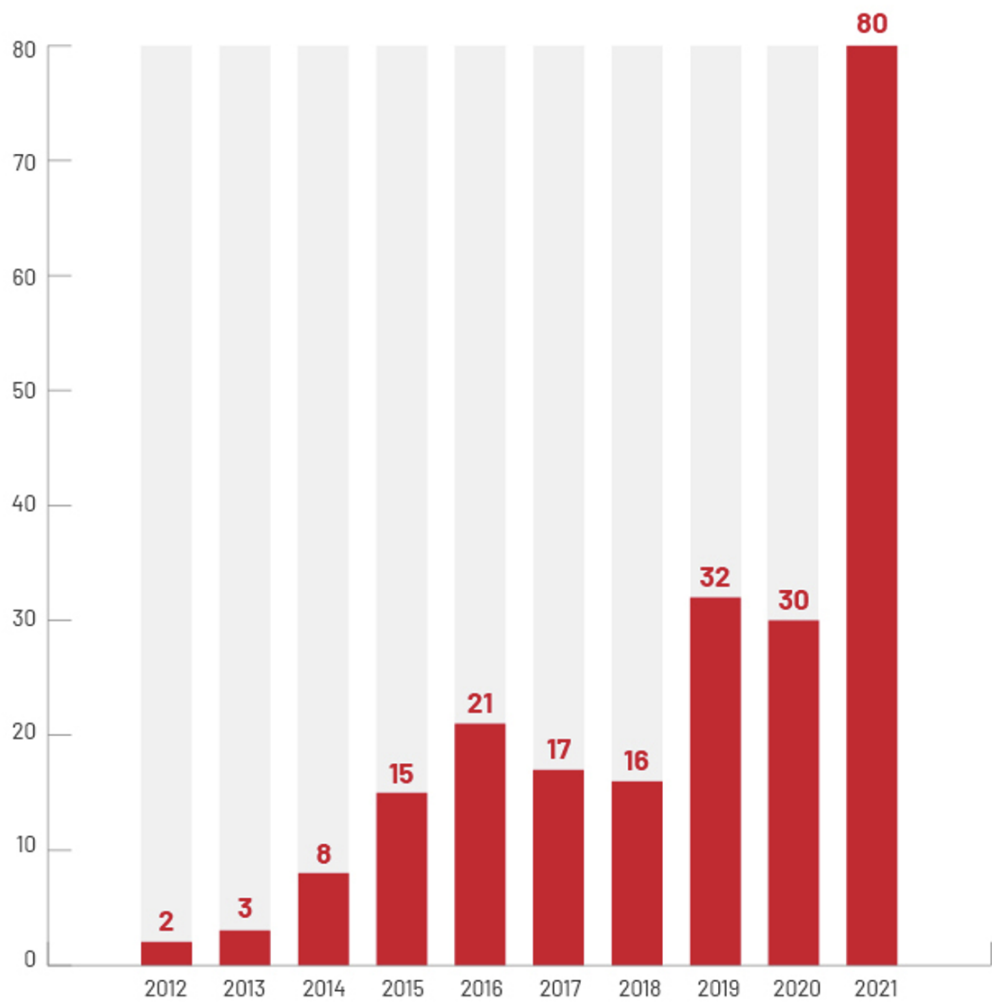
In many cases, the vulnerabilities are exploited in a very sophisticated way and there is no other choice but to take the affected services and servers offline, but in other cases the while the vulnerability itself can stem from very intricate details in the software, the exploit itself can be quite simple (for example a well defined series of packets).

When we deal with the latter case then we might be able to quickly create an in-network application to subvert possible attacks while we wait for the verified patch of the vulnerable software. This method has the potential of saving a lot of down-time and money.

2.5 Network security applications

Based on the reasons discussed in section 2.4 and because of the flexibility and speed offered by in-network solutions, many applications originally implemented to run on user-devices or servers have been ported to network devices.

Zero-Days Exploited 2012-2021



MANDIANT

Figure 2.4: Number of zero-days exploited by year. [13]

Using these applications in-network can in many cases boost the performance achievable with them and place them naturally in a way that fits the data manipulation, collection or filtering done by them much more than their original place of execution.

2.5.1 Packet filtering(P4Guard)

In many cases, network exploits are simple enough so that they can be identified by comparing the values of possibly malicious packets to stored signatures. Creating and deploying an in-network hotfix for an exploit can be imperative for a lot of reasons. A trustworthy patch for the vulnerable software might not exist yet or first we might need to make sure that installing it wouldn't create problems on its own. Another motivating circumstance might be that the network might not be totally under our control, hence we wouldn't be able to make changes to the vulnerable device or service. P4Guard is a protocol-independent and platform-agnostic software-based firewall that has been designed to tackle such security use cases.

2.5.2 OS fingerprinting (P40f)

Operating System fingerprinting can be an extremely useful tool for network administrators. Knowing precisely how many instances of certain operating systems run in our network can enable them to deploy intricate network policies and offer deeper insight into problems behind user complaints (e.g., some software that is part of an OS might conflict with a service running the network causing some hard to trace-back behaviour).

P0f monitors the network's TCP/IP header information and assigns a likely operating system to each stream based on the stored signatures. All this is done passively, without sending probe packets and without consuming network bandwidth.

P40f presented in [14] is a P4 implementation of P0f. P40f fingerprints the traffic passing through it by examining TCP/IP headers, relying heavily on TCP options. The design allows specific and generic rules to exist as signatures, while still providing a degree of flexibility in configuration. This flexibility is enhanced by the idea of fuzzy matching, which means that if some key fields match, P40f (and p0f too) can loosen the examination on other fields. When P40f matches an operating system label to a packet, it can perform three actions predefined by the user:

- Drop the packet

- Drop the packet in question and all subsequent packets from the same source IP
- Redirect the packet to a previously specified IP address

P40f randomly selects a number of HTTP GET packets that the control plane analyzes. The reason for this is the fact that these are too complex to be processed on a commodity switch.

2.5.3 Port Knocking (P4Knocking)

Port knocking is a well established security solution, usually deployed in order to obstruct attackers in port scanning a server and mapping out what services are available. In its most basic form the honest actor first has to knock (send one or more packet(s) to the agreed upon destination port(s)) to gain access to the internal network.

The knocking part can be as simple as requiring the user to send a single UDP or TCP packet to the right port, but it's not unseen to make this process much more complex, anything from requiring a sequence of packets that target a static sequence of destination ports to requiring a sequence that targets dynamically calculated ports or even a sequence that takes into consideration the times between packets.

With the use of P4RROT, our main focus is to create flexible building blocks that enables network administrators to be creative and easily deploy solutions that fit the requirements of their networks. We believe that, based on our work, anyone with a sufficient oversight of the technologies involved can create their own knocking mechanism with relatively very little work.

As we have already pointed out, port knocking can take on a variety of forms and this flexibility allows it to be useful in a number of scenarios. A system administrator working in a school might has to do some work in an IT lab, here a solution to grant him/her access beyond the default access level in the lab might be to include the performance of a port knocking in the administrator's scripts that need this higher level of access.

Port knocking can also work as a last resort against reverse shells and malware 'talking home'. This is of course stipulating that the malicious code running on our network was written without exact knowledge of the security measures or that even if there is a malicious actor observing the network traffic we can create dynamic sequences that look random enough to minimize the likelihood of giving away a complete solution.

Another scenario we would like to highlight and that is making the access to the network resources a bit more intricate is the deployment of port knocking solutions in data

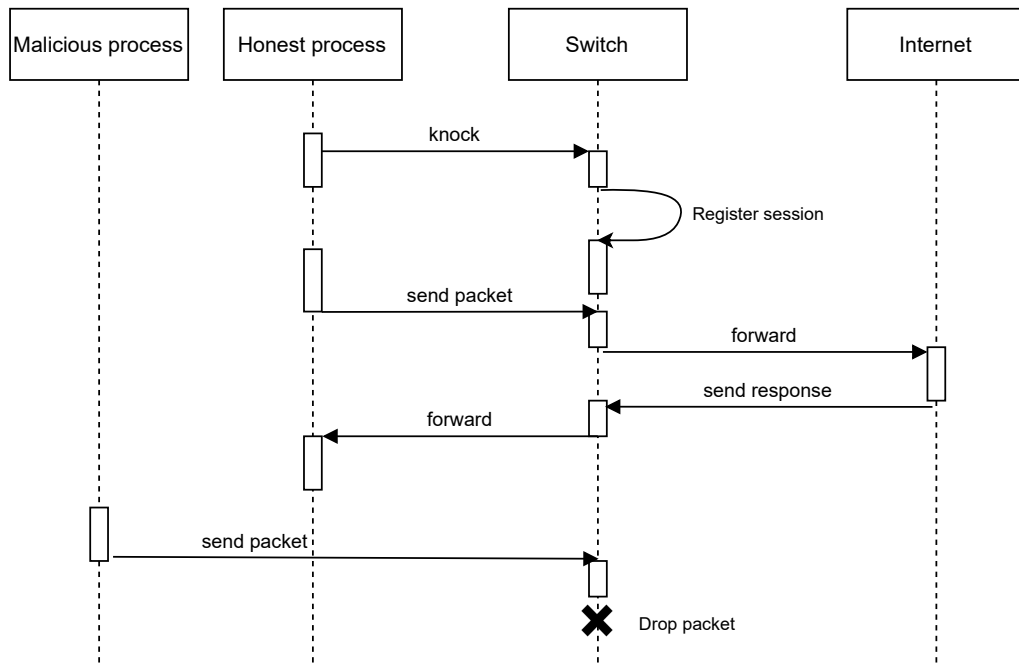


Figure 2.5: Using port knocking inside the network

centers. Here the knocking mechanism can be used to gate-keep the access for example to higher than average speed links without requiring a central solution.

All of the above described use-cases differ slightly from the classical use of port knocking in the way we orient and place our service, as it can be seen on Figure 2.5. We think it's important to note that the core technical implementation (the knocking mechanism itself) stays the same, the new part of the idea is the environment we surround it with.

Chapter 3

Method

In this chapter we describe how we approached the task of taking intricate in-network security applications and reproducing them in P4RROT, meanwhile keeping our eyes on the real goal of unearthing recurring patterns that can be incorporated into P4RROT as a generally reusable command or other construct. We also describe most such findings (the ones that we deem most attention worthy) kind of taken out of the context of the exact application we have developed them for, since the main point behind creating commands and other reusable structures is so they can be building blocks for many applications. To further enunciate this point, we show examples of how the created commands/constructs can be used to solve typical security and non-security problems.

3.1 Extending P4RROT

Throughout this work our method of choice was to reproduce the applications introduced in Section 2.5 incrementally. Meaning, to map out the application's logic, check whether any existing P4RROT commands can be incorporated and what commands will have to be implemented and then write the high-level code first. After seeing the ideal P4RROT version of the application (methods that are not necessarily implemented, but their names present a good idea of what they should do) we thought about how these commands will be the most flexible and fit into the already existing framework that is P4RROT.

3.2 CheckControlPlaneSet command

As we have described in the introduction an important property of SDN is the separation of the Data Plane and the Control Plane. The Data Plane handles the packets (does the dirty work if you like) while the Control Plane makes the higher level decisions for example which policy to apply. On the technical level this can manifest as the Control Plane populating the match-action tables of the Data Plane.

In many applications, security and non-security alike, we want to check whether a concrete value falls into a category defined by us. For example, in a security application we might want to apply an IP-address whitelist/blacklist or check whether a certain combination of (or a single) value(s) in packets corresponds to a known exploit. As a non-security example, we can think of the process of learning MAC addresses on a local network, here checking whether we already know an address follows the same pattern.

To eliminate the need for writing code for this pattern again and again, we created the CheckControlPlaneSet P4RROT command that creates a match-action table, that can be modified dynamically from the Control Plane and two actions that set a chosen field or variable to True or False based on whether the checked value is considered as an element of the category we wish to represent. The default behaviour of this command is to check all given keys with exact matching, which means that the stored and the checked values have to be entirely the same. However, just as P4 supports range and ternary matching, we support them as well, each key given to the command can be checked differently. Ternary matching means that we supply a mask for each entry and every time the table checks whether a value fits the entry it performs a logical AND operation before the comparison. Figure 3.2 is the P4RROT code of a simple application, where all we do is check whether an IP address is on whitelisted or not. Figure 3.1 depicts the P4 code generated by P4RROT.

3.3 ReadFromControlPlaneSet command

The ReadFromControlPlaneSet command is very similar to the CheckControlPlaneSet command detailed in Section 3.2. The only difference between them is that while the latter writes a single Boolean value into a variable or field, the former can write any built in type into an arbitrary number of variables or fields.

P4

```
BOOL_T allowed;  
action setter_action_uid4_true() {  
    allowed = 1;  
}  
action setter_action_uid4_false() {  
    allowed = 0;  
}  
table exact_match_table_uid3 {  
    key = {  
        hdr.genhdr_uid1.src_ip: exact;  
    }  
    actions = {  
        setter_action_uid4_true;  
        setter_action_uid4_false;  
    }  
    size = 1;  
    const default_action = setter_action_uid4_false;  
}
```

Figure 3.1: Simple Whitelist example in P4

P4RRROT

```
(  
fp  
    .add(CheckControlPlaneSet(["src_ip"], "allowed"))  
)
```

Figure 3.2: Simple Whitelist example in P4RRROT

```

P4RROT
(
fp
.add(ReadFromControlPlaneSet([{"name":"hdr.ipv4.src", "match_type":"ternary"},
{"name":"hdr.tcp.dstPort", "match_type":"exact"}],
["should_redirect", "redirect_to", "security_category"])))
)

```

Figure 3.3: Packet categorization example in P4RROT

```

P4
action setter_action_uid4(B00L_T paramuid5,bit<32> paramuid6,bit<8> paramuid7) {
    should_redirect = paramuid5;
    redirect_to = paramuid6;
    security_category = paramuid7;
}
table control_plane_set_table_uid3 {
    key = {
        hdr.ipv4.src: ternary;
        hdr.tcp.dstPort: exact;
    }
    actions = {
        setter_action_uid4;
    }
    size = 1;
}

```

Figure 3.4: Packet categorization example in P4

Our motivation for the creation of `ReadFromControlPlaneSet` was the many many cases where we need to get some additional data stored on the device, or choose one of many categories data a value can be a part of. The simplest of examples can be an application where we would like to forward frames based on their destination MAC address. Here we will need the port the frame has to exit the switch as additional information. A bit more complex case can be a more fine-grained categorization of packets based on for example their source and destination IP address and their destination TCP/UDP port. With this construction we can easily restrict who can access what services in our network, set a security level and get additional settings to inform further processing.

Figure 3.3 depicts a P4RROT code snippet, where we use the `ReadFromControlPlaneSet` command to pair a security level and decide whether to redirect (and if yes, where to) a TCP packet, based on its source IP address and destination TCP port. Figure 3.4 depicts the P4 code generated by the P4RROT code snippet.

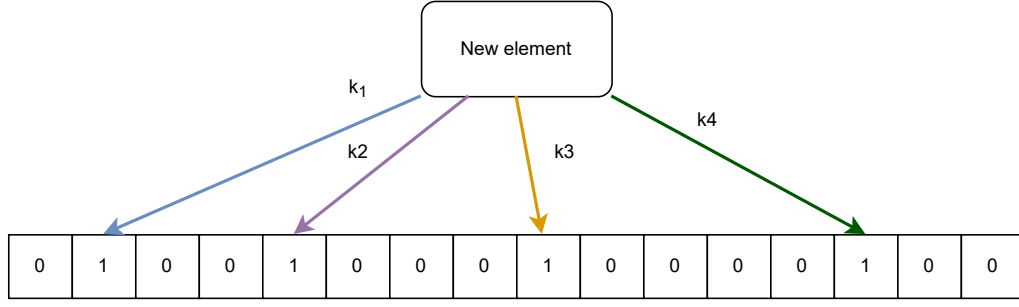


Figure 3.5: How a Bloom filter works

3.4 GeneralBloomFilter data structure

A Bloom filter is a probabilistic data structure representing a set. It is used to test whether an element is a member of a set or not (in other words, it provides membership queries). As a trade off for efficient space utilization we can get false positive results, but not false negative results.

Bloom filters represent their state as an array of m bits. When we would like to add a new element, we hash it (the new element) k times with ideally k hash functions, each hashing resulting in a bit position. The bits on the resulting positions are set to 1. Figure 3.5 depicts a simple example of adding a new element, where $m = 16$ and we have $k = 4$ hash functions. To check whether an element is in the Bloom filter, we hash it k times, just in the case of adding a new element, but here, instead of setting the bits, we check whether all of them contain 1, if they do, then we say that the element is in the set.

According to [15] Bloom Filters historically have a lot of applications in the networking world, mainly because the conscious treatment of the possible false positives is heavily outweighed by how efficiently they use space. They categorize their use into four big categories:

- Creating content summaries for collaboration in peer-to-peer and overlay networks.
- In probabilistic resource locating algorithms.
- To speed up or simplify routing protocols.
- To create data summaries in routers and other network devices.

[15] shows the probability of a false positive result in a Bloom filter is approximately:

$$(1 - e^{-kn/m})^k$$

```

P4RROT
(
fp
.add(If("interesting"))
.add(GeneralPutIntoBloom("addressesbloom", ["hdr.ipv4.src", "hdr.ipv4.dst"]))
)

```

Figure 3.6: Bloom filter snippet in P4RROT

```

P4
if (interesting==1){
    hash(_addressesbloom_hash_1uid2, HashAlgorithm.crc16, (bit<32>) 0, {hdr.ipv4.src, hdr.ipv4.dst}, 32w15);
    hash(_addressesbloom_hash_2uid3, HashAlgorithm.csum16, (bit<32>) 0, {hdr.ipv4.src, hdr.ipv4.dst}, 32w15);
    _addressesbloom_result_1uid4 = _addressesbloom_result_1uid4 + 1;
    _addressesbloom_register_1.write(_addressesbloom_result_1uid4, _addressesbloom_hash_1uid2);
    _addressesbloom_result_2uid5 = _addressesbloom_result_2uid5 + 1;
    _addressesbloom_register_2.write(_addressesbloom_result_2uid5, _addressesbloom_hash_2uid3);
}

```

Figure 3.7: Bloom filter snippet in P4

Where k is the number of hash functions (or hashing rounds), n is the number of possible elements we can put in the set and m is the number of bits comprising the filter.

As a part of this work we have reimplemented Bloom filters in P4RROT. The already existing implementation wasn't capable of accepting elements that are composed of multiple primitive types (this is why the name of our implementation is GeneralBloomFilter) and used a single hash function for all rounds. Our implementation can work with complex elements and uses the two hash functions (CRC16 and CSUM16) we can find in most P4 capable devices. We do realize that the latter might be less than ideal, but at the time of writing this thesis, the support of modern hash functions is a work to be done in programmable network devices, fortunately, what we have is enough for our practical purposes regarding Bloom filters.

Our implementation uses the built-in hash functions of the Behavioral Model (in the framework for this work, we implemented the filter for this target) and registers to represent the bit arrays. We provide two commands to interact with the filters, the GeneralPutIntoBloom and the MaybeContains commands. With the use of the former we can add elements to the set, while with the latter we can query the membership of an element.

Figure 3.6 depicts a simple P4RROT code snippet, in which we store every source, destination IP pair that we deem interesting. Figure 3.7 depicts the resulting P4 code snippet.

3.5 Register commands

In P4 the most common way to retain state between packets is to use registers. This is because it doesn't require direct communication with the Control Plane, which is an extremely high-cost operation. Retaining state can come handy in many situations. For example let's imagine a situation in which we would like to store the last n values seen from a sensor connected to the network device running our code. Consulting the Control Plane each time would be a solution, but it would result in a very slow application and that kind of strips away the appeal of solving the problem in-network. Instead of that, we can maintain a register storing a single value (a Shared Variable), that has an initial value of 0 and a register storing n values (a Shared Array). For every new sensor value, we can overwrite one of the old values using the value in the Shared Variable as an index which we then increment. If the stored index value equals n , then we can set it to 0 again. This way, we created a simple and efficient solution.

However useful they are, dealing with registers in P4, can be quite difficult, some targets (for example the Intel Tofino series of programmable switches) put restrictions on how many times they can be read/written per packet or how complex the register actions can be. To give a feel for these restrictions, our sensor value storing algorithm would be completely implementable on some targets but not on a Tofino switch (because of the restrictions on how many times we can read/write the registers). For these devices we would need to form a sort of queue of registers. We would define an order between n register instances and with every execution write the new value into the first register, at the same time reading its old value. Then, we could write the old value into the second register and so on.

This circumstance, the fact that they are very useful combined with the amount of code we need to write to use them really motivated us to create commands that increase the average P4RROT user's quality of life when it comes to registers. We added the following commands:

- **ReadSharedArray**: Reads the value stored on the given index.
- **WriteSharedArray**: Writes the given value onto the given index.
- **IncrementSharedArray**: Increments the numeric value stored on the given index with given amount. Possibly returns the result.

```

P4RRROT
fp.add(
IncrementSharedArray("counter_register", uint32_t, uint32_t,
"sensor_value_category", read_new_to = "counter_value", env=fp.get_env())
)

```

Figure 3.8: Register code snippet in P4RRROT

```

P4
RegisterAction<bit<32>,bit<32>,bit<32>>(counter_register) increment_shared_array_action_uid2 = {
void apply(inout bit<32> to_store, out bit<32> to_return){
    to_store = to_store + 1;
    to_return = to_store;
}
};

```

Figure 3.9: Register code snippet in P4

- **DecrementSharedArray**: Decrements the numeric value stored on the given index with the given amount. Possibly returns the result.
- **XORSharedArray**: Performs an exclusive or operation on the numeric value stored on the given index with the given value. Possibly returns the result.

Figure 3.8 depicts a P4RRROT code snippet that stores in a register how many instances were seen from each sensor value category. Figure 3.9 depicts the P4 declaration generated by this snippet.

3.6 Comparison table commands

Comparing values with each other is essential in many applications, be it in-network or other realms of programming. Checking whether a dynamic value is above a given threshold or whether a dynamic value equals some prespecified constant. A simple in-network example can be a case where we need to decide whether amount of packets sent by a node in a given time frame exceeds the allowed amount or not.

As we have already mentioned in Chapter 1 and Section 3.5 the different P4 targets can have different restrictions. One, very descriptive example of this is how on some targets we are not able to directly compare arbitrarily big values (here, big meaning values like integers in other languages). A trick to work around this is perform a subtraction on the values we would like to compare and then examine the result with the use of a match-action table. Knowing that this solution will work is not a given for beginners and even if we know the trick, coding it again and again can be quite tedious besides offering up a lot of opportunities for mistyping something. Hence, we created a number of Comparison commands that implement this method:

```
(
fp
.add(GreaterThanTable("too_much", "packets_received", "MAX_AMOUNT_ALLOWED"))
.add(If("too_much"))
    .add(DropPacket())
)
```

Figure 3.10: Compare table code snippet in P4RROT

```
P4
```

```
action setter_action_uid4_true() {  
    hdr.genhdr_uid1.too_much = 1;  
}  
action setter_action_uid4_false() {  
    hdr.genhdr_uid1.too_much = 0;  
}  
table eval_table_uid2 {  
    key = {  
        difference_variable_uid3: ternary;  
    }  
    actions = {  
        setter_action_uid4_true;  
        setter_action_uid4_false;  
    }  
    size = 2;  
    const default_action = setter_action_uid4_true;  
    const entries = {  
        0b000000000000000000000000000000000000000000000000000 : setter_action_uid4_false();  
        0b000000000000000000000000000000000000000000000000000 : setter_action_uid4_false();  
    }  
}
```

Figure 3.11: Compare table declaration code snippet in P4

- **GreaterThanTable:** Subtracts the second given operand from the first one and return a bool based on the examination of the result with a match-action table. Checks whether the first operand is greater than the second.
- **SmallerThanTable:** Subtracts the first given operand from the second one and return a bool based on the examination of the result with a match-action table. Checks whether the first operand is smaller than the second.
- **EqualTable:** Subtracts the second given operand from the first one and return a bool based on the examination of the result with a match-action table. Checks whether the given operands are equal.
- **NotEqualTable:** Subtracts the second given operand from the first one and return a bool based on the examination of the result with a match-action table. Checks whether the given operands are not equal.

Figure 3.10 depicts a P4RROT code snippet that decides whether the value in the `packets_received` variable is greater than the one in the `MAX_AMOUNT_ALLOWED` constant, if yes, then it drops the packet. Figure 3.11 depicts the P4 declarations generated by the snippet and Figure 3.12 depicts the P4 code snippet in the application's `apply` block.

```

P4
if (hdr.genhdr_uid1.isValid()){
    #define OUTPUT_HEADER_SIZE 9
    difference_variable_uid3 = hdr.genhdr_uid1.packets_received - hdr.genhdr_uid1.MAX_AMOUNT_ALLOWED;
    eval_table_uid2.apply();
    if (hdr.genhdr_uid1.too_much==1){
        ig_dprsr_md.drop_ctl = 0x1;
    }
    #undef OUTPUT_HEADER_SIZE
}

```

Figure 3.12: Compare table apply code snippet in P4

```

P4RROT
(
fp
.add(Digest(["hdr.ipv4.src","hdr.udp.srcPort", "payload"], ["needs_control_plane"]))
)

```

Figure 3.13: Digest code snippet in P4RROT

3.7 Digest command

In Chapter 1 we described the differences between the Data Plane and the Control Plane. The Data Plane deals with the actual packets, while the Control Plane deals with the higher level logic. The former is usually programmable in P4, while the latter offers more than one option (Python and C for example), at least in the cases that we are interested in in this work.

The Data Plane is normally much faster and highly optimized, while the Control Plane is slower but has a lot less restrictions, hence the communication between them is always a high-cost operation, but in some cases a necessary one. For example, if we have a security application designed to detect intrusions, then there we might accept the high-cost of the synchronization between the two planes so that the Data Plane can ask the Control Plane to send a summary of a severe intrusion attempt to a central server. A non-security related example can be the case of an application that aggregates sensor values in the network but if it runs into a packet that it can't process, then it delegates the task to the Control Plane. In both cases, the event that triggers the sending of a digest (mechanism for the Data Plane to send a message to the Control Plane) is rare enough so that its high time consumption is acceptable.

A digest is a mechanism for the Data Plane to send a message to the Control Plane. In our work we created the Digest command that takes a list of values we would like to send and a list of keys. It creates a match-action table using the given keys, based on which the user can insert entries that decide whether the situation really is worthy of sending a digest or the packet should be simply dropped. It also injects the appropriate declarations, checks and metadata setting lines into all necessary places throughout the generated code so that the digest can be sent when really needed.

```

P4
action setter_action_uid8() {
    ig_dprsr_md.digest_type = 2;
}
action drop_packet() {
    ig_dprsr_md.drop_ctl = 0x1;
}
table set_digest_or_drop_table_uid7 {
    key = {
        hdr.genhdr_uid1.needs_control_plane: exact;
    }
    actions = {
        setter_action_uid8;
        drop_packet;
    }
    size = 1;
    const default_action = drop_packet;
}

```

Figure 3.14: Declaration generated by the digest code snippet

```

P4
Digest<generated_digest_uid6>() generated_digest_uid6;

```

Figure 3.15: Deparser declaration generated by the digest code snippet

Figure 3.13 depicts a P4RROT code snippet, where the application sends a digest containing the source IP address, the source UDP port and the payload of a UDP packet if the situation is deemed complex enough to require the attention of the Control Plane. Figure 3.14, 3.15 and 3.16 depict the declarations, the declaration for the deparser and the apply code snippet generated by the P4RROT snippet respectively.

3.8 Smaller commands

Throughout our work we created a number of "smaller" commands. Commands that map a single P4RROT line to a single or a few P4 line(s). The motivation behind these was either, to have a shorthand for frequently used but trivial command combinations or to provide continuity, so that a few single lines don't have to be placed by hand after the

```

P4
if (ig_dprsr_md.digest_type==2){
    generated_digest_uid6.pack({hdr.ipv4.src,hdr.udp.srcPort,hdr.genhdr_uid1.payload});
}

```

Figure 3.16: Deparser apply code generated by the digest code snippet

code generation. These commands might have constituted an effort to create in a manner that wouldn't disrupt P4RROT's flexible structure, but we don't deem them important enough to give them their own section here.

The created smaller commands are the following:

- **DropPacket:** Sets the appropriate metadata value so that the packet currently in the pipeline will get dropped.
- **LogicalNot:** Take a variable or field containing a Boolean value and perform the logical not operation on it.
- **LogicalOr:** Take a variable or field containing a Boolean value and perform the logical not operation on it.
- **AssignConst:** Assign a constant value to the given variable or field.
- **MaskedNotEqualsConst:** Applies a logical and operation between the given value and the given mask, then compares the result with the third given value.
- **GetPacketLength:** Reads the length of the current packet into the given variable or field.
- **LeftShift:** Performs a left bit shift operation on the first given value using the second given value as how many places to shift with.
- **RightShift:** Performs a right bit shift operation on the first given value using the second given value as how many places to shift with.
- **ClonePacket:** Sets the appropriate metadata value so that the current packet will be cloned by the device.

Chapter 4

Recreating in-network security applications with the use of reusable commands and templates

In this chapter, we describe how we were able to reproduce three in-network security applications with the use of P4RROT and introduce our idea for a construction that provides transitioning between application states in a consistency ensuring manner.

4.1 Hotfixes

4.1.1 NTP vulnerabilities

Network Time Protocol (from here on referred to as NTP) in its current (at the time of writing this paper) is quite safe against attacks outside the network when it is configured well, but the same thing can't be said about its outdated versions that are still in use on a surprisingly high number of servers.

For example, there is a known vulnerability detailed in CVE-2015-8138 that was resolved in NTP 4.2.8p6, that allowed attackers to successfully forge NTP response packets that could set the time on the client's side to an arbitrary value. In 2019, three years after the publication of the vulnerability more than 2000 publicly available DNS resolvers ran a version of NTP that was susceptible to the mentioned time-shifting exploits according to [16].

Payload	Expected result
Exploit	Drop
Regular	Allow

Table 4.1: NTPGuard tests

4.1.2 NTPGuard

As an example we created NTPGuard for the Tofino Native Architecture, a really simple application written with the use of P4RROT, which is meant to block packets carrying malicious payloads aiming to exploit one of NTP’s well-known vulnerabilities described in Subsection 4.1.1. This application is comparable to the ones created with P4Guard. P4Guard checks whether a packet classifies as an exploit with the use of our ‘CheckControlPlaneSet’ command introduced in Section 3.2. The implementation is so simple that we were able to include its whole body on Figure 2.2.

4.1.3 Tests and results

To test our implementations we defined a number of test-cases, created the appropriate Control Plane rules and then probed the configured application with UDP packets. The cases we tested for are listed in Table 4.1.

4.2 P40f

We described P40f in Section 2.5.2. It is an in-network Operating System fingerprinting application. It monitors the packets’ TCP/IP header information and assigns a likely operating system to each stream based on the stored signatures. All this is done passively, without sending probe packets and without consuming network bandwidth. P40f was by far the biggest application we chose to reproduce. It was initially introduced as an application for the Behavioural Model, then later implemented to run on Tofino as well ([14]). Our work is based on the first implementation.

To extract the TCP header size P40f uses binary search, the reproduce this we used the binary shift commands introduced in Section 3.8. Since the signatures used by P40f are really granular we needed to perform equation checks on smaller fields. To do so, we used the ‘MaskedEqualsConst’ and ‘MaskedNotEqualsConst’ commands which we also introduced in Section 3.8. One of P40f’s main features is mainting statistics about what

Operating Systems it has seen on the network. This is done with the help of registers. In the implementation of this feature we made good use of our register commands introduced in Section 3.5. P40f randomly selects a number of HTTP GET packets that the control plane analyzes. The reason for this is the fact that these are too complex to be processed on a commodity switch but the user-agent field they contain can be used for OS identification. To keep track of these flows and avoid analyzing the same flow multiple times P40f uses a Bloom filter. To reproduce this part we used the reimplemented Bloom filter construction described in Section 3.4.

4.2.1 Parsing complex structures

Describing how we would like to parse packets is something we can do in P4RRROT, but only to a degree. Since in most cases this step is pretty straight forward, there is no need for the extraction of metadata, recursion or other tricks, P4RRROT has a really simple way to handle parsing. However parsing TCP with all its possible options is not part of the mentioned most cases, hence we introduced a new template to the P4RRROT ecosystem that handles this task for us. Most of this template is adapted from an open-source project with licenses it under the Apache License, Version 2.0. The appropriate references and legal text is included in the P4RRROT repository.

4.2.2 Tests and results

To test our p40f implementation we compiled our solution and the original one with the p4c compiler and ran them on the BMv2 software switch. Since the signatures and packets used in the original P40f paper weren't at our disposal we couldn't exactly reproduce the evaluation steps described there, but we did compare the results given by the original and our implementation on part of the CIC-IDS2017 dataset. In this comparison the labels generated by the two implementations were exactly the same. During the testing we made sure that all parsed TCP fields get exercised. Based on this, we claim that we managed to reproduce P40f for BMv2 with the use of P4RRROT quite adequately.

4.3 P4Knocking

We created a lightweight version of the Hybrid control and data plane offloading solution for port knocking presented in [17] for the Tofino Native Architecture. The original

implementation expects a sequence of knocking packets, while our simplified version only uses one packet. Our solution works with UDP knocking packets instead of TCP to simplify parsing, but we believe this modification does not affect the security or the usability. To implement this solution we used the 'Digest' command presented in Section 3.7 in order to 'register' the new sessions identified by their source IP address, source UDP port and destination UDP port. To decide whether a session has been cleared already can be easily done with the help of our 'CheckControlPlaneSet' command introduced in Section 3.2 which might fortify our point that we have found a nontrivial but frequently emerging building block of in-network applications.

4.3.1 A more complex version

In order to take advantage of the advantages described in Subsection 2.5.3 we created a more complex replica of P4Knocking, requesting a sequence of packages from users. In the original P4Knocking implementation, registers indexed by the source IP address are used to store counter values indicating how far the user has completed the knocking sequence, while we use the state machine approach described below.

In this extended application, the central question was how to represent and update the sequence of states in a way that ensures consistency and allows a large number of sessions to exist simultaneously.

In this specific task, we define an arbitrary start and accept state represented by bit strings stored in registers. The transitions between the states are defined as a mapping between the destination ports of the incoming UDP packets and the bit string, the user combines (the exact action used in the transition is predefined, but can be easily changed at the code level) with the current state. Mapping between target port and bit string is done with the 'ReadFromControlPlaneSet' P4RRROT command presented in Section 3.3, which targets the common problem of mapping values to each other. To do so it generates a match-action table using exact matching (in this particular case, otherwise the command is capable of handling ternary matching as well) in which the pairs are stored as entries, this way making the mapping completely modifiable from the control plane.

To ensure state consistency, we needed to make sure that reading the current state and calculating the next state is done in a single atomic step. For this purpose, we used the P4RRROT commands 'XORSharedArray', 'ANDSharedArray' and 'ORSharedArray' briefly described in Section 3.8, which create register operations that can perform a logical

Port	Expected result
Incorrect	Deny
Correct	Allow

Table 4.2: Simple knocking tests

Required knocks	Order	Operation	Duplicate knock	Expected result
1	-	XOR	Yes	Deny
1	-	XOR	No	Allow
3	Correct	XOR	No	Allow
3	Correct	XOR	Yes	Deny
3	Incorrect	XOR	Yes	Deny
3	Incorrect	XOR	No	Deny
1	-	AND	Yes	Deny
1	-	AND	No	Allow
3	Correct	AND	No	Allow
3	Correct	AND	Yes	Deny
3	Incorrect	AND	Yes	Deny
3	Incorrect	AND	No	Deny
1	-	OR	Yes	Deny
1	-	OR	No	Allow
3	-	OR	No	Allow
3	-	OR	Yes	Allow

Table 4.3: Complex knocking tests

exclusive or, logical and, and logical or operation operations respectively on the bit string stored in the register, and the bit string requested as part of the transition. The register actions then store it in the register and return the result.

4.3.2 Tests and results

To test our implementations we defined a number of test-cases, created the appropriate Control Plane rules and then probed the configured application with UDP packets. The cases we tested for the simple version are listed in Table 4.2 and the cases we tested for the complex version are listed in Table 4.3.

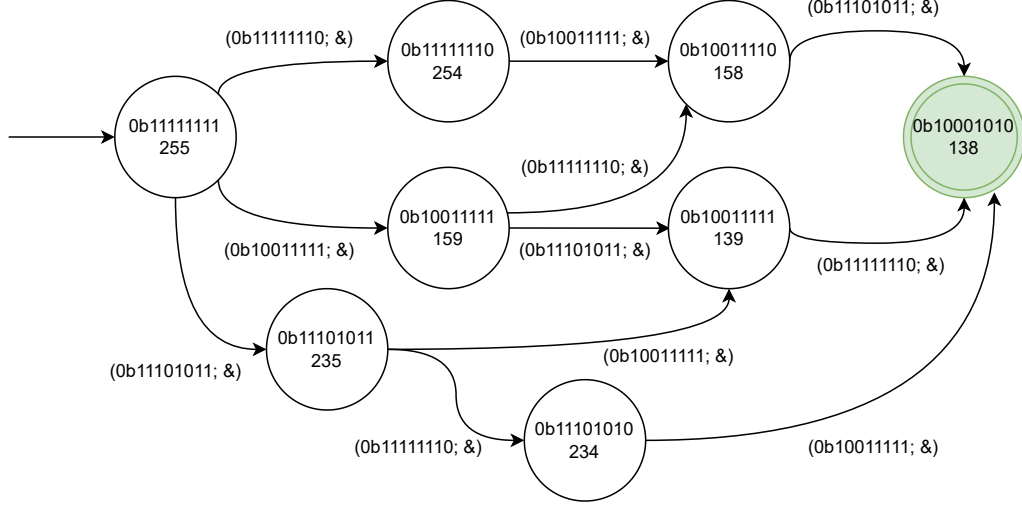


Figure 4.1: Single Touch State Machine

4.4 Single touch state machines

While working on the implementation of P4Knocking (described in Section 4.3) we noticed that the problem of transitioning safely between states in an application while guaranteeing consistency is a rather non-trivial and frequently occurring one in the in-network world.

We found that abstracting the problem into a state machine operating on bit strings is a really promising idea. Here transitions are possible actions on the stored state. Each transition consists of a read operation and performs a simple logical operation (and, or, xor) on the bit string and stores the result with the constraint that they must be atomic to ensure consistency. We would like to introduce the term 'One-touch state machine' for this general solution.

As we have seen in Section 4.3 this solution worked perfectly well when we needed to store how far along we are in a port knocking sequence. We believe that the further analysis of this construction can yield some interesting results. A lot of restrictions can be formulated based on scenarios found in practical application needs, hence we plan on investigating it further, beyond the frame of this work.

Figure 4.1 depicts an example of our idea, here 255 is the starting state (signified with the arrow without a starting point) and 138 is the single accepting state, signified by the double outline and the color green.

Chapter 5

Evaluation

5.1 Functional correctness

We defined and executed test-cases for NTPGuard and our port knocking applications (we described the details in Subsection 4.1.3 and Subsection 4.3.2), which all passed. We strongly believe that we managed to test all equivalence classes regarding the possible inputs of each application. For P40f we validated our implementation (we described the details in Subsection 4.2.2) with the use of the original one on a vast amount network traffic, which gives us confidence in its correctness.

5.2 Evaluating the code generation

Table 5.1 compares the length of source code files for different solutions of the investigated use cases. The rows of the table include: 1) P4RROT code: the Python code using

Observed code	P4Knock	P4Guard	P40f
P4RROT code	35	38	187
Generated P4	93	56	356
Complete generated	309	272	768
Original implementation	381	-	884

Table 5.1: Comparison of source code lengths

Resource	P4Knock	Extended P4Knock	NTPGuard
Stages	25%	25%	16.7%
SRAM	1.3%	2.0%	0.4%
TCAM	0%	0%	0%
ALU	0%	2.1%	0%

Table 5.2: Used resources on Tofino

P4RROT; 2) Generated P4: the P4 code generated directly by P4RROT; 3) Complete generated: The P4 code generated by P4RROT and the template created by hand; 4) Original implementation: The code published by the original authors of the reproduced application; Blank lines are excluded from the count, while comments are included.

We observed that in cases where a comparison can be made, both the Python code generating the solution and the generated P4 files are shorter than the original implementation. Naturally, we don't want to claim that code length is a good measure of any quality of an implementation. We believe this comparison shows that using the highly reusable template provided in P4RROT and a well-thought-out design in mind, it is faster to write an order of magnitude less Python code than to write P4 code directly.

Of course, there may be cases where the exact operation or construct required to solve a particular problem is not available in P4RROT and would be too much work to implement. Even in these cases we can presumably produce P4 code base quickly, which can be modified manually. We believe this mindset aligns with P4RROT's goal of being a support tool rather than a replacement for P4 programming.

5.3 Performance and resource usage

Table 5.2 summarises the resource usage of the applications implemented by us using the Tofino Native Architecture. We used SDE 9.4.0 to compile the P4 code.

We observe that the fact that we used P4RROT to create our implementations didn't come with an unreasonably high cost, since we have used well under 50% of all kinds of resources, leaving space for potential further functions. We regret the fact that a direct comparison wasn't possible with the original implementations due to a number of reasons. For P4Guard we couldn't find a publicly available code base and the other applications weren't implemented with the Tofino Native Architecture. P4of does have an implemen-

tation for it, but as a basis for our work, we used the original version that was made to run on the BMv2 software switch.

Chapter 6

Summary

6.1 General conclusions

We found P4RROT well suited to support network-security use cases, and we successfully reproduced the solution of 3 prior works. While admitting that comparing number of lines is not a good measurement of quality we still think that our results (summarized in Table 5.1) strengthen the point that using P4RROT’s highly reusable commands can speed up the implementation of P4 applications.

Although P4RROT’s design is not created for non-whole byte length special fields and flags, we showed that it is able to accommodate them. During our work, we contributed multiple missing features along with support for the Tofino Native Architecture (TNA). Most of our work is available in P4RROT’s repository under open-source licensing.

6.2 Future work

Thanks to the focus on higher-level abstraction provided by P4RROT, it is much easier to quickly experiment with new ideas. Such example of new ideas is the Single Touch State Machine (STSM) construction described in Section 4.4. With the use of P4RROT we were able to quickly build it into our solution. Further exploring STSMs seems very promising, since it’s a general solution to the problem of keeping track of state, while reliably maintaining consistency. In the future we would like to evaluate the prospect of creating disaggregated STSMs, employing STSMs in other use cases and the advantages of further mathematical constraints on the abstract model.

In this work, we focused on security use cases, and it will probably remain a constantly changing area need for fast patches and quick adaptation providing numerous similar projects. However, P4RROT does not focus exclusively on this area, rather it aims to facilitate high-level application layer logic and general computing. One of the other potential areas for exploration is natural science projects dealing with high-throughput network traffic.

6.3 Acknowledgments

I'm extremely grateful to my thesis supervisors Csaba Györgyi and Sándor Laki for their patience and continuous feedback.

I'm also grateful to my friends and family for their encouragement and patience in this tumultuous period of time.

Bibliography

- [1] " E. Haleplidis Ed." et al. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 4726. RFC Editor, Jan. 2015. URL: <https://www.rfc-editor.org/rfc/rfc7426.txt>.
- [2] P4 Language Consortium. *P4 Workflow*. 2022. URL: <https://p4.org/> (visited on 11/23/2022).
- [3] Eder Ollora Zaballa and Zifan Zhou. "Graph-To-P4: A P4 boilerplate code generator for parse graphs". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–2.
- [4] Theo Jepsen et al. "Forwarding and routing with packet subscriptions". In: *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 2020, pp. 282–294.
- [5] Jonathan Vestin, Andreas Kassler, and Johan Åkerberg. "FastReact: In-network control and caching for industrial control networks using programmable data planes". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2018, pp. 219–226.
- [6] Csaba Györgyi et al. "NETREACT: Distributed Event Detection in Sensor Data Streams with Disaggregated Packet Processing Pipelines". In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. IEEE. 2022, pp. 249–253.
- [7] John Sonchack et al. "Lucid: a language for control in the data plane". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 731–747.
- [8] Vikram Nathan et al. "Demonstration of the marple system for network performance monitoring". In: *Proceedings of the SIGCOMM Posters and Demos*. 2017, pp. 57–59.

- [9] Srinivas Narayana et al. “Language-Directed Hardware Design for Network Performance Monitoring”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 85–98. ISBN: 9781450346535. DOI: 10 . 1145 / 3098822 . 3098829. URL: [https : / / doi . org / 10 . 1145 / 3098822 . 3098829](https://doi.org/10.1145/3098822.3098829).
- [10] Arpit Gupta et al. “Sonata: Query-driven streaming network telemetry”. In: *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 2018, pp. 357–371.
- [11] Qiao Kang et al. “Programmable {In-Network} Security for Context-aware {BYOD} Policies”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 595–612.
- [12] IBM LLC. *How much does a data breach cost in 2022*. 2022. URL: [https://www . ibm . com / security / data - breach](https://www.ibm.com/security/data-breach) (visited on 09/19/2022).
- [13] James Sadowski. *How much does a data breach cost in 2022*. 2022. URL: [https : / / www . mandiant . com / resources / blog / zero - days - exploited - 2021](https://www.mandiant.com/resources/blog/zero-days-exploited-2021) (visited on 11/23/2022).
- [14] Sherry Bai, Hyojoon Kim, and Jennifer Rexford. “Passive OS Fingerprinting on Commodity Switches”. In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 2022, pp. 264–268. DOI: 10 . 1109 / NetSoft54395 . 2022 . 9844109.
- [15] Andrei Broder and Michael Mitzenmacher. “Network Applications of Bloom Filters: A Survey”. In: *Internet Mathematics* 1.4 (2003), pp. 485–509. DOI: [im / 1109191032](https://doi.org/10.1109/191032). URL: <https://doi.org/>.
- [16] Philipp Jeitner, Haya Shulman, and Michael Waidner. “The Impact of DNS Insecurity on Time”. In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 266–277. DOI: 10 . 1109 / DSN48063 . 2020 . 00043.
- [17] Eder Ollora Zaballa et al. “P4Knocking: Offloading host-based firewall functionalities to the network”. In: *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 2020, pp. 7–12. DOI: 10 . 1109 / ICIN48450 . 2020 . 9059298.

List of Figures

1.1	Example of the P4 workflow [2]	5
2.1	The high level overview of P4RROT's usage.	7
2.2	Example P4RROT script	7
2.3	Example P4 code	8
2.4	Number of zero-days exploited by year. [13]	10
2.5	Using port knocking inside the network	13
3.1	Simple Whitelist example in P4	16
3.2	Simple Whitelist example in P4RROT	16
3.3	Packet categorization example in P4RROT	17
3.4	Packet categorization example in P4	17
3.5	How a Bloom filter works	18
3.6	Bloom filter snippet in P4RROT	19
3.7	Bloom filter snippet in P4	19
3.8	Register code snippet in P4RROT	21
3.9	Register code snippet in P4	21
3.10	Compare table code snippet in P4RROT	22
3.11	Compare table declaration code snippet in P4	22
3.12	Compare table apply code snippet in P4	23
3.13	Digest code snippet in P4RROT	23
3.14	Declaration generated by the digest code snippet	24
3.15	Deparser declaration generated by the digest code snippet	24
3.16	Deparser apply code generated by the digest code snippet	24
4.1	Single Touch State Machine	31

List of Tables

4.1	NTPGuard tests	27
4.2	Simple knocking tests	30
4.3	Complex knocking tests	30
5.1	Comparison of source code lengths	32
5.2	Used resources on Tofino	33

Date of last changes made to the content of this work:2022.12