Can we exploit buggy P4 programs?

Mihai Valentin Dumitru, Dragos Dumitrescu, Costin Raiciu
University Politehnica of Bucharest
firstname.lastname@cs.pub.ro

ABSTRACT

Recent verification works have found numerous bugs in P4 programs. While it is obvious bugs are undesirable, it is currently not known what effects these bugs have in practice? In this paper we take a first look at the potential of exploitation for such bugs: we first examine how three different targets behave when unspecified behaviours are triggered, finding a range of potentially exploitable behaviours; we use these to attack two concrete programs. We find that the security impact of such exploits can be high, but that the severity of the attack depends on the target.

CCS CONCEPTS

• **Security and privacy → Network security**;

ACM Reference Format:
Mihai Valentin Dumitru, Dragos Dumitrescu, Costin Raiciu. 2020. Can we exploit buggy P4 programs?. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373360.3380836

1 INTRODUCTION

Following the trend towards network programmability, languages such as P4 [1] or Broadcom's NPL [2] allow programming dataplanes that can run at line rate in production networks, disproving the decades-old mantra that the network is either flexible or fast. Programmable switch deployments have already started, and are expected to continue: Barefoot's Tofino switches are used in production, and Tofino2 will be shipping soon; Broadcom recently announced Trident 4, a new programmable chip (shipping in 2020).

Making dataplanes easily programmable enables unprecedented network flexibility, but it may come at the cost of robustness. A recent array of verification works has examined existing P4 programs [3]–[6], finding many categories of bugs in most of them. A common bug is to access invalid headers (headers which the current packet does not have, but which are syntactically in scope), but many others exist. Verification work can pinpoint such bugs as long as the programmer specifies a snapshot of the table rules [4] or invariants that will be obeyed by the table rules [3]. With this feedback, it should be relatively simple for programmers to remove bugs from P4 programs. Unfortunately, eliminating bugs may be more complicated in practice. Such tools require significant input from the programmer and a degree of verification expertise, and are still research prototypes.

In this paper we take a first step at understanding to what extent these bugs are exploitable in practice. We first code a series of simple, buggy programs to observe the actual behavior of targets, in those cases where the standard leaves it undefined. We tested three targets: the bmv2 software switch, P4-NetFPGA and Barefoot's Tofino switch. Then we examine a simple NAT implementation in P4 to see if it contains any exploitable bugs, as well as as switch.p4, the most complex P4 program publicly available today. We find that exploiting these two P4 programs is possible by powerful attackers (that know the program, the table rules and are directly attached to the switch) and that the exploit depends on the target. We then discuss the implications of our findings to network security.

2 BUGS IN P4 PROGRAMS

P4 enables programmers to specify how packets should be processed by a switch. A P4 program has a few parts, as shown in Figure 1:

- A parser which dictates how the packet is transformed from bits into headers. The parser specifies all possible header combinations the P4 program will accept. After the parser executes, the header fields are accessible as "global" variables (where each header is a structure containing several fields) which the rest of the program can access.
- Match-action tables than can match on arbitrary header fields and packet metadata, and that can execute user-defined actions upon a match. Actions can modify header fields or metadata, can add or remove headers and decide the packet's fate (e.g. clone, drop, forward).
- A control block which specifies how the packet will be matched against the tables; typically there are two control blocks, one for the ingress and one for the egress pipeline.
- A traffic manager which handles buffering, schedules packets on egress ports, drops the ones which are marked for dropping, implements prioritization and available AQM algorithms. This component is given by the target and can be typically configured but not programmed.
- A deparser that specifies how the active headers are laid out on the wire (if not provided, the parser is also used for this purpose).

A P4 architecture defines how these parts are connected and the interfaces between them. A P4 target is a device which supports one or more architectures and can run programs written for those architectures. Example architectures include simple switch and simple router (implemented by bmv2), Tofino Network Architecture (TNA) supported by the Tofino switch and Portable Switch Architecture (or PSA, supported by multiple targets including bmv2 and Tofino).

It is important to note that the P4 program only partially specifies

SOSR '20, March 3, 2020, San Jose, CA, USA

ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380836

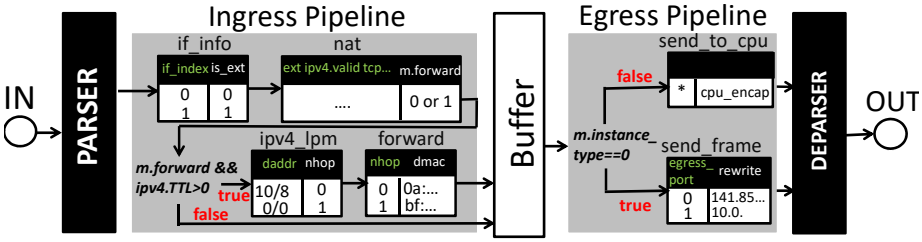

Figure 1: P4 program example: simple NAT

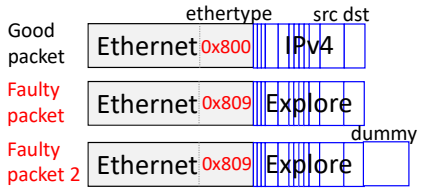

Figure 2: Packets used to explore target behaviour.

the packet-processing functionality: the full functionality also includes the table entries provided at runtime by a controller running on the switch or in a centralized location. The table entries specify which packets will match and what action should be applied.

In the NAT example, packets first match the `if_info` table which sets metadata that tracks the packet's ingress interface and can decide to drop the packet. Next, all packets match the `nat` table which decides to forward the packet in case of a hit, or send it to the controller otherwise. The packets that must be forwarded and have positive TTL match the `lpm` and `forward` tables before they reach buffering. After buffering, packets are either sent to the controller or on the wire, using the `send_to_cpu` and `send_frame` tables.

Prior verification work [3]–[6] has identified *classes of bugs* that P4 programs can exhibit. We are interested in those bugs which involve undefined behaviors, such as:

- Accessing fields from invalid headers (headers that are not valid for the current packet).
- Reading fields from uninitialized headers or metadata.
- Out-of-bounds header stack accesses.

And bugs which involve architecture-specific behaviors:

- The possibility to create infinite loops, where the same packet is repeatedly resent to the beginning of the pipeline.
- Unintentionally processing dropped packets.
- Implicit forwarding decision for certain packets.

Existing verfication works can in principle find all instances of such bugs, even in complex P4 programs, but to do so they require the programmers to carefully describe the type of table rules that might be inserted in the program at runtime; for p4v [3] and p4-assert [6] this is a controller specification (700 line specfication for the switch.p4 program), while for Vera [4] this is a representative snapshot of the table entries. Both approaches require a lot of programmer effort, but do identify all bugs using this effort; it remains to be seen if such techniques are adopted.

Consider this code snippet taken from our simple NAT:

```
if (meta.do_forward == 1 and ipv4.ttl > 0){
   apply(ipv4_lpm);
   apply(forward);    }
```

This code assumes the ipv4 packet is valid for the current header at this point in the program. With certain values in the NAT tables, this assumption is incorrect: a packet without an IP header can reach this point, and the target will read the `ttl` from the invalid ipv4 header. Furthermore, the `ipv4_lpm` matches on the IP destination address, resulting in an unpredictable forwarding decision.

We are interested in understanding what would happen in practice if such programs were deployed. The ideal behaviour would be for the switch to raise an exception: the packet is ejected from the pipeline and the controller is informed of the exception; this is the equivalent of a hardware trap in commodity processors when unmapped memory is accesed. Exceptions, however, are very expensive to implement in hardware; we thus expect that such packets will be processed, and the effects of writing or reading invalid headers will depend on the target. We explore target dependent behaviours in detail next.

3 BUGS ON REAL TARGETS

To understand the effects of P4 bugs on real targets, we wrote simple P4-14 and P4-16 programs and ran them on the three most popular P4 targets today:

- BMV2 software switch with the simple_switch [7] target is a favourite for early development and simple testing, however it is not meant for production deployment. It offers all the functionality of a P4 target, but its performance is subpar (30Kpps [8]).
- P4-NetFPGA [9] toolchain that compiles P4 programs into bitfiles that can be run on the NetFPGA SUME board (four ports at 10Gbps) [10].
- The Barefoot Tofino switch is the only production-grade programmable switch today; it's also fast, with up to 65 ports running at 100Gbps.

For the bmv2 target, we use a virtualized environment where we deploy a P4 program, inject packets with Scapy and explore the returning packets with tcpdump. For the Tofino, we deploy the program and then use one connected host to generate packets (with Scapy) and examine the results. For NetFPGA, we used P4-NetFPGA to produce bitfiles and deployed them on the Sume board, as well as in the cycle-accurate Vivado simulator.

3.1 Reading invalid headers

We wrote a program that executes an action for all incoming packets which copies the IPv4 header checksum over the Ethernet ethertype field (both 2B wide), and sends the packet back out on the same interface.

We then sent several Ethernet/IPv4 packets to the target followed by a packet containing only the Ethernet header, which triggered the target to read the invalid IPv4 header checksum field. [1]. All packets were padded to 64 bytes (the minimum Ethernet frame size) with arbitrary bytes. We then examined the contents of the ethertype field in the Ethernet-only packet as it left the target.

[1] We term as faulty such fault-triggering packets

Both the P4-NetFPGA and the Tofino returned packets with ethertype of 0, suggesting that the invalid IPv4 header access yielded value 0. The bmv2 simple_switch target, however, returned packets with ethertype set to the value of the previous Ethernet/IPv4 packets, thus exfiltrating data from previous packets that were forwarded by the switch.

To confirm these results, we ran another test where we defined one header called *Explore* which has the same fields as IPv4. *Explore* follows Ethernet when the ethertype is 0x809, as shown in Figure 2; *Explore* and IPv4 are mutually exclusive with regards to the parser. The P4 program will simply send IPv4 packets through unchanged; explore packets, however, are faulty and trigger an action that copies over all but one of the fields of the invalid IPv4 header into the fields of the valid *Explore* header; only the protocol number is set to a constant that does not exist in the IPv4 header.

We again injected multiple IPv4 packets followed by one explore packet and checked the resulting *Explore* header. For the bmv2 target, this header echoes values from the previous IPv4 packets, resulting in a reliable data exfiltration attack from the switch. For the P4-NetFPGA, the *Explore* is all zeros except the protocol field, consistent with the previous results.

For Tofino, the behaviour is different: the explore header is unchanged, except for the protocol field which is set to the expected value. This behaviour happens with or without a check for ipv4 not valid before applying the copy action. There are two possible explanations for this behaviour: the first is that actions which involved invalid headers are not executed on the Tofino; this however contradicts our previous findings where reads return 0. Another explanation is that the compiler maps the two mutually exclusive headers (explore and ipv4) at the same starting address, therefore the ipv4.tll and explore.ttl field are stored at the same memory location, making the copying a no-op. Further experiments seem to confirm this hypothesis.

So far, we only read invalid headers in actions. Is the behaviour similar when an invalid header is read in the control block? To find out, we used a program which tests all possible values of the IPv4 flags field and triggers a unique action for each value:

```
if (ipv4.flags==0)      apply(flags0);
else if (ipv4.flags==1) apply(flags1);
else if (ipv4.flags==2) apply(flags2);...
```

Each flagsY table has a single action which is applied to all packets, setting the ethertype field to 0x80Y. As previously, we send Ethernet/IPv4 packets followed by one Ethernet/Explore packet and examine the value of the ethertype of the return packet to understand what value is read by the control block.

We find that the bmv2 target reads the flags value from previous IPv4 packets, the P4-NetFPGA reads 0 and the Tofino reads the value of the Explore flags field, further confirming that the compiler overlaps these headers in memory.

We wondered if table lookups using invalid headers yield the same result. We used a single table with as many actions as possible values for the IPv4 flags field, where actionY set the value of ethertype to 0x80Y, as above. The behaviour of P4-NetFPGA and the Tofino is consistent with our previous observations (0 and Y read from the explore field), however the bmv2 target behaved differently: 0 was matched in the table, not the values of previous IPv4 headers as seen in previous tests; this seems to be because of a special case in the bmv2 code that clears the key before lookup.

Finally, we explored what concrete values are obtained in other cases which are left unspecified by the standard (e.g. uninitialized variables). We found that all targets, even bmv2, yield sane defaults such as 0 or "the first defined value" (for `enum` and `error` types). Our findings are summarized in Table 1.

3.2 Writing invalid headers

To gain further insight on target behaviour with invalid headers, we wrote a program that sets all the fields of the invalid IPv4 header to predefined constants, then checks if any of these writes has succeeded; if so, the ethertype field is changed to a predefined value. The P4_16 standard explicitly allows writes to invalid headers to modify *undefined* state in the system (section 8.21).

Both the bmv2 simple_switch and the P4-NetFPGA targets execute the write to the invalid header and then the read succeeds, implying that the invalid header lives in its own memory area, and the write is not conditioned on the header being valid.

Tofino also executes the writes and reads the written value; however, writing to the invalid IPv4 header results in changes to the Explore header. This is because the header layout in memory can vary quite a lot for similar headers. For instance, if we change the Explore header by adding one dummy field at the end (see Figure 2), IPv4 and Explore either do not overlap anymore, or partially overlap, depending on the size of the dummy field.

3.3 Loops

To understand looping behaviour, our aim was to understand what effects infinitely looping packets have on other packets in the pipeline. We coded a P4 program that behaved differently depending on the type of packet received: it always looped one type of packet infinitely (using one of the recirculation options available), it looped another type of packet just once, and finally allowed other packet types through. The SimpleSumeSwitch architecture used on P4-NetFPGA does not support recirculation, so we do not test loops there.

The behaviour also depends on how loops are implemented. In case of the resubmit primitive, we observed that in BMV2, infinitely resubmitted packets essentially block the ingress pipeline, dropping all subsequent packets. However, due to hardware and safety reasons, on Tofino a packet may only get resubmitted once; it gets dropped if resubmitted again.

Another way to implement loops is to use egress to egress clone operations. We created a program that clones IP packets on the egress pipeline to egress and also retains packet metadata. Our results indicate that both BMV2 and Tofino infinitely loop packets through the egress pipeline effectively sending a clone to the configured egress port, and also looping the packet to the beginning of the egress pipeline. This way, a single ICMP echo request resulted in (millions of) packets being continuously sent to the destination.

As creating infinite loops is possible on some targets, programmers need to take care in avoiding them whenever they employ recirculation primitives. P4 architectures have special metadata fields whose values indicate the path taken by a packet (normal,

		BMV2	NetFPGA	Tofino
read from invalid header	action	previous valid value	0	0
	control	previous valid value	0	0
	table key	0	0	0
read from uninitialized	header	0	0	0
	variable	0	0	0
	out param	0	0	0
write to invalid header		changes undefined state	changes undefined state	changes undefined state
out-of-bounds header stack access		runtime indexes unsupported	runtime indexes unsupported	runtime indexes unsupported
infinite loop	resubmit	blocks processing	unsupported	max one resubmit
	e2e clone	destination flooding	unsupported	destination flooding
ressurecting packets		set egress port	set egress port	set egrees port and mark undropped
implicit forwarding		port 0	drop	drop

Table 1: Concrete behaviors on tested targets

resubmitted, recirculated etc.). These can be used to avoid resubmitting a packet more than once. Alternatively, one can implement a TTL field in user-defined metadata (conserved by resubmission) and write the checks and changes necessary to manage this field, a recommendation that can also be found in the PSA documentation.

3.4 Resurrecting dead packets

Dropping a packet simply involves changing some metadata to mark the packet as dropped, without stopping its processing: it is simply too expensive to eject the packet from the pipeline. The dropped packet will then continue through the pipeline until buffering where it should be dropped; however, the drop decision may be accidentally cancelled, resurrecting the packet. In P4-14, the function to mark the packet for dropping was confusingly called *drop*, which could trick programmers into assuming the packet gets dropped straight away; P4-16 leaves dropping mechanisms completely architecture-specific.

Certain architectures make it easier to accidentally resurrect packets marked for dropping. The `v1model` architecture is such a case because it uses the standard metadata field `egress_spec` both for unicast routing and drop marking. When a packet is marked to be dropped, the target sets the `egress_spec` metadata to a predefined value (511). When a dropped packet triggers an action that sets the `egress_spec` to the value of an egress port, that packet will be revived instead of being dropped.

The Portable Switch Architecture uses a special metadata field to mark dropped packets. Thus, the only way to resurrect a previously dropped packet is to set the egress port and also unmark the packet for drop. The latter action makes the programmer's intent unambiguous. It is clear that he did not *accidentally*, but rather *intentionally*, revive a packet.

The SimpleSumeSwitch architecture also uses a special metadata field to mark dropped packets, but the field is currently marked "deprecated"; the documentation explicitly indicates that dropping should be marked by setting the `dst_port` metadata to 0, making it easy to resurrect packets.

3.5 Implicit forwarding behaviour

What happens to a packet whose output port was not set? Is it dropped or does it get forwarded in a target specific manner? Our exploration shows that in simple_switch, whenever the egress_spec metadata is unset at the end of the ingress pipeline, the packet is sent out to port 0. This is dangerous, since, under the right circumstances, a malicious user could flood clients connected to port 0 with traffic of its own choosing, while at the same time bypassing ACLs.

As far as hardware targets are concerned, we observe that the Tofino safely defaults to dropping a packet which doesn't specify a valid egress port. P4-NetFPGA also defaults to dropping a packet, which is consistent with the fact that uninitialized fields are set to zero (including the `dst_port` metadata); zero is not a valid port as P4-NetFPGA uses a one-hot encoding for ports.

4 EXPLOITING P4 PROGRAMS

Our experiments have revealed how multiple targets implement behaviors left undefined (or architecture-defined) by the P4 standard and they hint to possible exploits against such targets. In this section we provide an initial exploration of possible attacks and their effects for generic programs, as well as looking in detail at two P4 programs.

We note that bugs in P4 programs are reminiscent of similar bugs in general-purpose programming languages such as C: reads from invalid headers are similar to use-after-free attacks and writing invalid headers are similar to writes through dangling pointers. However, despite many similarities between P4 and C, the attacks possible on P4 targets are inherently weaker than those possible on buggy C programs for multiple reasons:

(1) While the C standard states that the behaviour of programs that read or write invalid memory locations is undefined, the P4 standard is more restrictive: only the resulting value is undefined (for reads) or the location of the write (for writes), not the behaviour of the program after the fault.

(2) The P4 code is immutable once deployed, so classical code-injection attacks have no equivalent in the P4 world.

(3) The P4 program cannot directly control the next instruction to be executed, since the control flow is immutable; there is no equivalent of a "jmp addr" or a "ret addr" in the P4 world. This means that code reuse attacks, such as Return Oriented Programming attacks [11], [12] are not possible and that control-flow integrity is provided automatically [13].

Most attacks against P4 programs can be classified as Data Flow Integrity attacks [14] where the attacker can read or write data that the programmer had not intended. We explore these after we discuss possible attackers.

Attacker model. There are two dimensions which determine the strength of an attacker: target knowledge and connectivity.

The strongest P4 attackers have *full knowledge*: they know the *P4 program* deployed on the target, the *target type*, have information about *intrinsic metadata* and know the *currently configured table entries*. This is the strongest attacker possible; weaker attackers include those that know only the P4 program and target type but not the table entries, those that do not even know the target, etc (partial knowledge).

Another dimension is how the attacker is connected to the target. The strongest attacker is directly connected to the target (*direct*), which means it can inject packets with exotic headers, including at layer 2. The second strongest attacker is in the same *LAN* with the target, and it can create exotic headers at layer 3 and above, for instance our Explore header. An even weaker attacker is one that lives in another *internal* network, being forced to create well-formed IP packets to get them to the target, but unencumbered by Internet ACLs. The weakest attacker is one in the *public* Internet, and it can reach the target only subject to checks from various stateful firewalls (e.g. that allow only outbound connectivity).

Next, we assume that the attacker has full knowledge of the target and discuss where the attack can be mounted from; weaker attacker types are subject of future work.

4.1 Possible attacks on buggy P4 programs

Based on the concrete behaviors of the tested targets, we employ a STRIDE [15] analysis to group possible threats for a P4 dataplane. STRIDE is a widely used security model consisting of six threat categories designed to cover most attacks observable in practice.

Spoofing. Reading arbitrary values (either from previous packets, zeros or from the same packet at different offsets), resurrecting dropped packets and implicit packet processing behaviour can help the attacker bypass ACLs the P4 program is trying to enforce. An attacker can craft a packet whose provenance (ingress port, layer 2/3 host etc.) will be mistaken or overwritten by the program, such that later sections of it interpret it differently.

Tampering. All targets keep some global state in registers which is updated conditionally on the values in the packet. If the program has bugs, the attacker can exploit these to pollute such global state, making it unreliable — for instance, dropped packets counted by the ACL before they are resurrected by the attack would affect the integrity of such logging mechanisms, which could confuse the network operator and make the attack go undetected for longer.

Repudiation. With regards to a programmable dataplane as our considered system, we do not see how repudiation threats are applicable.

Information disclosure. Possibly the worst attack against P4 programs is snooping on other traffic by forcing the switch to read from invalid headers. Buggy tunneling code is particularly vulnerable as the outer header is copied to the inner header; if the outer header is invalid yet the tunneling action is executed, the inner header may still be emitted.

A BMV2 simple_switch target running buggy programs is vulnerable to this threat, as the invalid reads will return values from previous packets, allowing an attacker to snoop on other traffic going through the switch. When a vulnerability exists, mounting this attack locally and in the LAN is feasible, from the internal network it may be possible, and difficult from the public Internet. From our experience so far, snooping on other traffic with this approach is not possible for the Tofino or P4-NetFPGA targets.

Denial-of-service. A particularly disruptive attack is denial-of-service. By sending a single packet to the BMV2 target we can make it black-hole all other traffic (exploiting a resubmit loop). Sending an ICMP packet to a Tofino or BMV2 target running a vulnerable P4 program that clones on egress results in a stream of packet copies being emitted at output indefinitely; such an attack could bring down the target network / machine. This attack can be mounted by local, LAN, and internal attackers, and could even be mounted by public attackers under the right circumstances.

Elevation of privilege. The same factors that can lead to spoofing (reading arbitrary values, resurrecting dropped packets, implicit packet processing behavior) enable privilege escalation attacks, in which the attacker can send a specially crafted packet that could bypass security filters. While the severity and effectiveness of such attacks remain to be established, we show below that these are possible in two P4 programs that we have examined in more detail.

We now assume the strongest possible attacker (full knowledge, local) against a BMV2 target and explore concrete attacks that are possible in this scenario.

4.2 Attacking a simple NAT

Our first attack uses the Simple Nat code available as part of P4 tutorials. We also built a simple controller application which adds NAT table entries to previously unknown connections. We perform our attack against the simple_switch target of BMV2. The setup includes three hosts: an internal host (I), an external one (E) and a rogue host (R) (the attacker), all directly connected to the target.

Bypassing ACLs with revived packets. The first observation is that reviving dead packets may lead to security issues (such as ACL bypass). In this example, the `if_info` table maps ingress ports to their nature: internal or external and additionally serves as a per-interface ACL. The controller correctly sets up I as an internal host and E as an external one. Furthermore, the controller marks packets from R's interface to be dropped. The packet continues through the pipeline and encounters the NAT table. If the attacker guesses an existing NAT mapping, he can get his packet past the NAT table to the IPv4 routing table. Here, the egress_spec is overwritten and the packet is resurrected.

DoS. Another exploit is having the controller forcibly populate the `nat` table with entries of the attacker's choosing. Assume that R knows a class of IP addresses for which NAT is allowed. Then, he may send out TCP packets originating from distinct IP source/TCP source port into the NAT box. Note that the per-interface ACL should filter R's requests and no traffic should ever go out of the P4 program. Nevertheless, the drop rule in the `if_info` is equivalent to marking the packet for being dropped while at the same marking

the port with a default nature. In the p4-14 spec, metadata are implicitly assigned the value zero, which corresponds to an internal port. The NAT table cannot distinguish between a packet received from I and one received from R which is marked for dropping. Since R knows the IP addresses which may get natted, he can force a NAT miss and have its packet redirected to the controller. If the controller is "dumb" (as in our example), it will add a new entry to the table and allow the packet to pass. This is a DoS attack where the `NAT` table is flooded with connections of the attacker's choosing.

Privilege escalation. We have investigated a third kind of exploit, which leverages the special CPU header. In our example, a special kind of header was envisaged for communication between the CPU and the P4 dataplane. This header sits below the Ethernet header and is identified by a preamble of 64 zero bits. It contains several metadata fields; most importantly, the ingress_port. Whenever the P4 program receives such a packet, it assumes it came from the CPU and uses the information inside the header to overwrite the ingress port.

Armed with this information, R may forge such a packet and completely bypass all ACLs by simply pretending to have come from a different port. The egress pipeline will ensure that the CPU header gets popped and a correct IPv4 packet egresses the switch.

Programs that handle special packets which bypass most dataplane logic should include necessary checks for the origin of these packets, in order to prevent such attacks. The P4 program should be aware of a particular port meant for communication to and from the CPU and should reject any packet with a CPU header received from another port. The PSA architecture defines a special macro (`PSA_PORT_CPU`) for this port; v1model does not, but a programmer could make a definition of their own.

The previous tests show that even programs implementing simple tasks, such as simple_nat may be subject to security issues when the controller is "dumb" and that additional checks are needed to safeguard against such issues. We now turn our attention to a more complex and well engineered example, switch.p4.

4.3 Attacking `switch.p4`

`switch.p4` is the reference P4 dataplane program. It provides support for many commonly used network protocols and may serve as a data center ToR router implementation. For our tests, we use the open-source version of switch.p4[2] and its underlying drivers - switchAPI, switchSAI and switchlink.

We have set up a simple source IPv4 ACL rule which blocks traffic from a given address (A). We tried to break this ACL using IPv4/IPv6 packets sourced at A, but failed in doing so.

We give it to the clean levels of abstraction which sit on top of switch.p4 (switchAPI and switchSAI). Our observation is that, for complex dataplane programs, it is unfeasible to directly populate table entries without introducing bugs. Thus, abstraction helps preserve safety invariants while keeping flexibility and performance guarantees.

However we were able to perform a **privilege escalation** attack. We observe that, much like simple_nat, switch.p4 also has a header destined to ensure communication between the CPU and the P4 dataplane. If we forge such a header with the proper bypass flags and an internal IPv4 packet with source A, we manage to effectively bypass the ACL set up in switch.p4 and have the switch flood packets to the outside, thus breaking the ACL.

This happens because there is no extra sanitization step to ensure that packets bearing the CPU header are actually originated at the CPU. An attacker leveraging this observation would thus be able to bypass almost all checks enforced therein.

[2]https://github.com/p4lang/switch

5 RELATED WORK

While there is a wide body of literature discussing the security of traditional programming languages, there are very few works which look at the security of programmable dataplanes. Schmid et al. provide a high-level overview of security of programmable dataplanes [16], drawing a parallel to security in software-defined networks (e.g. Openflow). Our work is complementary as it provides a detailed analysis of the risks introduced by P4 and ways in which they may be exploited. Ang Chen et al. [17] look at functional correctness of P4 dataplanes, trying to infer the normal distribution of packets to code paths via symbolic execution and then use it to detect attacks that deviate from this behaviour. This class of DoS attacks is also complementary to our work as it does not use any bugs in the P4 dataplane.

6 DISCUSSION

As most software, programmable data planes can exhibit bugs which may be exploited by attackers. The most vulnerable target is the bmv2 software switch which leaks information from previous packets; P4-NetFPGA and Tofino do not leak such information, but may be vulnerable to other attacks. We acknowledge that the bmv2 software switch is meant as a quick means of P4 prototyping and is not expected to be deployed in production; nevertheless it is a popular target and such data leaks must be fixed.

Bugs that originate from accessing undefined values (such as invalid or uninitialized headers) are easy to prevent. There already are several [3], [4] existing solutions capable of automatically detecting such bugs.

From the target manufacturer point of view, it seems that some behavioral alternatives are better than others, with respect to the ease with which bugs could be exploited. For example, marking a separate metadata field for dropped packets (as is the case for Tofino) makes it hard to resurrect dead packets "by mistake", simply by setting the egress port.

Our attacks on simple_nat demonstrate how dataplane bugs, coupled with a lack of coordination between the dataplane and control plane lead to dataplane misuse and security issues. The well-engineered switch.p4 lacks this kind of miscoordination, but the absence of some necessary checks means it can still be exploited by a powerful attacker. Our experience shows the need for programs to always sanitize packets which may lead to privilege escalation and only allow them from trusted agents (such as the CPU).

Nevertheless, these attacks require very strong attackers and appear difficult to mount from the Internet. In future work we will scrutinize more programs and use weaker attackers to fully understand whether P4 poses meaningful security risks compared to fixed-function switches.

ACKNOWLEDGEMENTS

This work was funded by CORNET H2020, a research grant of the European Research Council (ERC StG, no. 758815).

REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, Jul. 2014.

[2] Broadcom, *NPL: Open, High-Level language for developing feature-rich solutions for programmable networking platforms*, 2019. [Online]. Available: https://nplang.org/.

[3] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soule, H. Wang, C. Cascaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of ACM SIGCOMM 2018*.

[4] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, Budapest, Hungary: ACM, 2018, pp. 518–532, ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230548. [Online]. Available: http://doi.acm.org/10.1145/3230543.3230548.

[5] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18, Los Angeles, CA, USA: ACM, 2018, 5:1–5:7, ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3185497. [Online]. Available: http://doi.acm.org/10.1145/3185467.3185497.

[6] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of p4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: ACM, 2018, pp. 73–85, ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281421. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281421.

[7] P. language consortium, *Designing your own switch target with bmv2*, 2019. [Online]. Available: https://github.com/p4lang/behavioral-model.

[8] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/olteanu.

[9] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4netfpga workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: ACM, 2019, pp. 1–9, ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293924. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293924.

[10] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014, ISSN: 1937-4143. DOI: 10.1109/MM.2014.61.

[11] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[12] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, Washington, DC, USA: IEEE Computer Society, 2014, pp. 227–242, ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.22. [Online]. Available: https://doi.org/10.1109/SP.2014.22.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05, Alexandria, VA, USA: ACM, 2005, pp. 340–353, ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102165. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165.

[14] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06, Seattle, Washington: USENIX Association, 2006, pp. 147–160, ISBN: 1-931971-47-1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298470.

[15] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack, "Threat modeling-uncover security design flaws using the stride approach," *MSDN Magazine-Louisville*, pp. 68–75, 2006.

[16] A.-A. Agape, M. C. Danceanu, R. R. Hansen, and S. Schmid, "Charting the security landscape of programmable dataplanes," *CoRR*, vol. abs/1807.00128, 2018. arXiv: 1807.00128. [Online]. Available: http://arxiv.org/abs/1807.00128.

[17] Q. Kang, J. Xing, and A. Chen, "Automated attack discovery in data plane systems," in *Workshop on Cyber Security Experimentation and Test*, 2019.