Towards automatic exploitation of programmable networks

Mihai-Valentin Dumitru
University Politehnica of Bucharest
Bucharest, Romania
mihai.dumitru2201@upb.ro

Dragos Dumitrescu
University Politehnica of Bucharest
Bucharest, Romania
dragos.dumitrescu@cs.pub.ro

Costin Raiciu
University Politehnica of Bucharest
Bucharest, Romania
costin.raiciu@cs.pub.ro

2024 IEEE 10th International Conference on Network Softwarization (NetSoft) | 979-8-3503-6958-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/NetSoft60951.2024.10588899

***Abstract*—P4 verification works have found numerous bugs in programs of various sizes. While existing tools are efficient in finding bugs such as invalid header accesses, little effort has been put in understanding the potential impact of these bugs against the network. In this paper, we investigate whether these bugs can expose security vulnerabilities similar to those studied extensively for commodity CPUs. This work presents the design and implementation of `HackP4` – a tool which makes use of static and dynamic analysis techniques to assess security properties of P4 dataplanes. `HackP4` discovers vulnerabilities in P4 programs and automatically generates security exploits if they exist; otherwise, it provides guarantees of their absence. We present the results of running `HackP4` against several P4 programs and show the kind of vulnerabilities it is able to capture. Finally, we discuss best practices for mitigating bugs and minimizing the impact of vulnerabilities.**

***Index Terms*—p4, security, automatic exploit generation**

I. 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 switches are available from all major vendors (Intel's Tofino and Tofino2, Broadcom's Trident 4, Mellanox's Spectrum 2) and being rolled out in production.

Making dataplanes easily programmable enables unprecedented network flexibility, but may come at the cost of robustness. A substantial number of verification works has examined existing P4 programs [3]–[7], finding multiple bugs in most of them, such as invalid header accesses. Whether these bugs can be turned into exploitable vulnerabilities is still an open question. This paper is the first attempt to understand the security impact of bugs in P4 dataplanes.

The security characteristics of dataplanes have been difficult to assess so far, with few studies available [8], [9]. This is because fixed-function dataplanes are essentially black-boxes, making it difficult to systematically test or verify for exploits. Programmable network devices have the advantage of being auditable in terms of security.

Existing work in P4 program verification reports unsafe behavior, but does not reason about the effects of these bugs at runtime; [10] documents the results of experiments designed to uncover how such bugs manifest on real targets, i.e. are the corresponding packets dropped or do they leave the switch? This helps us distinguish benign bugs from vulnerabilities.

Our goal is to automatically discover whether a bug is exploitable. To this end, we develop `HackP4`, a tool which uses both static and dynamic analysis techniques to find certain classes of vulnerabilities in buggy P4 programs. If vulnerabilities are found, `HackP4` is able to produce packets to automatically exploit them; otherwise, it provides guarantees about their absence.

We believe this tool is useful for P4 developers, as an extra step in the deployment pipeline. It can be used to harden dataplanes via bug triage, much like Automated Exploit Generation (AEG) for traditional software [11]–[15]: given multiple bugs and limited resources for fixing them, the ones that can actually be exploited by attackers should be fixed first. Every reported bug is accompanied by the concrete bytestring of a packet that triggers it, enabling manual exploration and filtering of false positives (e.g. bugs that depend on some table entries that the control plane can guarantee will not occur).

Additionally, with a little manual intervention, `HackP4` can be configured to verify arbitrary application-specific constraints (see §IV-A), such as "a packet with a source port X is never forwarded with destination port Y".

Our starting point consists of the existing verification works that can statically find bugs in P4 programs [3]–[7]. Such tools stop when finding a bug, but this is not how actual targets behave. The exploration in [10] serves as a reference for the way in which such bugs manifest on existing targets. To decide whether a bug is exploitable and to generate an exploit, our work bridges the gap from current state of the art, providing the following contributions:

- We build `HackP4`, the first P4 verification tool that models P4 undefined behavior on real targets.
- We show how `HackP4` can automatically build exploits for real-life programs with thousands of lines of code.
- Finally, we describe a set of attacks which demonstrate `HackP4`'s exploit-generation capabilities on P4 programs and provide a set of mitigation strategies.

II. Bugs and exploits in P4

P4 enables programmers to specify how packets should be processed by a switch. While P4 is relatively simple and its instructions have well-defined semantics [16], programs

written in it are not exempt from bugs – as evidenced by verification works such as [3]–[7].

The P4 standard defines the general language constructs syntactically and semantically, but there are also some behaviors that are intentionally not defined: architecture-specific behaviors and undefined behaviors.

In this paper, we use the term "bug" to refer to undefined behaviors and to some problematic architecture-specific behaviors.

Architecture-specific behaviors A P4 program must be written for a specific *switch architecture*, which specifies the programmable blocks and their interactions, available `externs`, mechanisms for dropping/forwarding/cloning packets etc.

For example, to forward a packet in the ingress block under the `v1model` architecture, one needs to set a metadata field to some particular 9-bit integer value. To drop a packet, the same field must be set to a special value. Combined with the fact that there is usually no "early return/exit" from a control block, it becomes possible for a forwarding decision to revert an earlier drop decision, effectively "resurrecting" the packet.

On the Tofino Native Architecture, a similar mechanism is used for forwarding. But dropping requires setting a separate flag, making accidental packet resurrection unlikely.

We are concerned with two possibly-problematic-behaviors on the `v1model` architecture (and any others that employ similar mechanisms): packet resurrection and implicit forwarding.

In short, packet resurrection refers to the cancelling of a drop decision further down the processing pipeline; implicit forwarding refers to what happens to the packet when no explicit forwarding or drop decision has been made.

Undefined behaviors Some instructions' behavior is explicitly mentioned as being undefined by the standard.

In this paper we focus on: undefined reads (from invalid headers or uninitialized variables), writes to invalid headers and out-of-bounds accesses to register arrays/header stacks.

Unlike the infamous concept of Undefined Behaviors (UB) in C, where "everything can happen", the effects in the P4 world are much tamer. A read from an invalid field is guaranteed to produce *some value of that type*; it can't crash the program, or change control flow etc.

Vulnerabilities It is possible that some bugs can occur without any **observable effect**; by this, we mean that no packet which leads to triggering the bug is forwarded, resubmitted or recirculated. We call these "benign bugs".

Bugs that can be triggered by packets that are then emitted by the switch, represent **vulnerabilities**, because the packets affected are then visible in the network and could have negative effects on it.

We use the term **exploit** to describe a packet whose processing triggers a bug and results in a packet being emitted.

The aim of `HackP4` is to filter out benign bugs and discover vulnerabilities, then automatically generate exploits that trigger them.

III. Real target behaviors

In this paper, we focus on two P4 targets:

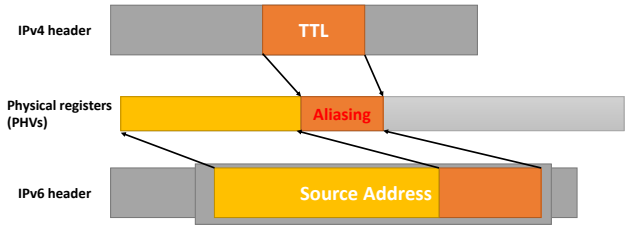

Fig. 1: A scenario in which the IPv4 TTL field overlaps with the last byte of the IPv6 source address.

- BMv2 software switch with the simple_switch [17] target, using the `v1model` architecture
- The Intel Tofino Switch [18] - the only production-grade P4 switch available today, using the Tofino Native Architecture (TNA)

Experiments in [10] show what are the actual, observable effects of triggering undefined behavior on real targets. We summarize here the results relevant for the design and implementation of `HackP4`, providing extra details obtained from additional experimentation.

A. Reading from invalid headers

One of the most common bugs discussed in P4 verification literature is accessing fields from invalid headers. The discovered behaviors are representative for the trade-offs performed by each target in order to optimize performance.

On Tofino, both header and metadata fields are zeroized before every packet; generally, reading from an invalid header yields a zero. However, a different value can be obtained from an invalid read, because no further zeroization takes place in cases where a header is invalidated during packet processing, or when a value is written to an invalid header (see the next section on invalid writes). Furthermore, headers that are mutually-exclusive from a parsing point-of-view can overlap in memory; reading from an invalid header can then produce a value from a valid, overlapping header. Assuming the scenario in Figure 1, reading the `ipv4.ttl` field of packets with an IPv6 header would yield the last byte of their source address.

On BMv2, header zeroization only occurs when fields are used as part of a table key; in other cases, an invalid header (i.e. that wasn't populated by the parser) will contain values from the last packet processed for which that header was valid.

B. Writing to invalid headers

The P4_16 standard [16] explicitly allows writes to invalid headers to modify *undefined* state in the system (section 8.25). As expected, both on BMv2 and on Tofino, writing a valid to an invalid header field will cause future invalid reads from that field to yield that value.

Because of the memory overlapping that can occur on Tofino between two mutually-exclusive headers, writes to an invalid header can modify *defined* state – fields, or parts of fields, of a valid, overlapping header. In the scenario of Figure 1, trying to decrement the `ipv4.ttl` field of an IPv6

packet would result in changing the source address of the packet.

C. Resurrecting dead packets

Dropping a packet involves changing some metadata to mark it as dropped; the processing doesn't stop, there is no early exit. The dropped packet will continue through the pipeline until buffering, where it should be dropped; however, the drop decision may be cancelled, resurrecting the packet.

The precise dropping mechanism is architecture-specific, with some architectures making it easier to accidentally resurrect packets marked for dropping. `v1model` uses the same metadata field both for unicast routing and drop marking. When a packet is marked to be dropped, the target sets that field to a special drop value. When a dropped packet triggers an action that sets the metadata field to a valid port ID, that packet will be revived instead of being dropped.

However, TNA (and the Portable Switch Architecture) uses a separate flag to mark dropped packets. 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 they did not *accidentally*, but rather *intentionally*, revive the packet.

D. Implicit forwarding behavior

The mechanism of forwarding a packet is also architecture-dependent; so is the behavior resulting from not explicitly taking a forwarding decision. The packet could be dropped or forwarded in some particular manner.

On the `v1model`, whenever the `egress_spec` metadata is unset at the end of the ingress pipeline, this is equivalent to forwarding the packet on "port 0", which can be a valid port identifier. 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.

On TNA, packets with no explicit forwarding decision are dropped.

IV. Implementing HackP4

`HackP4` consists of a static verification component and an enumerative packet generator.

To implement the first component, we choose as starting point an existing verification tool, `bf4` [7]. We inherit its bounded model checking approach and extend it by enhancing its instrumentation step to model target behaviors, as well as equipping it with a packet generation algorithm. Figure 2 shows the complete flow of `HackP4`.

`bf4` itself is implemented as a `p4c` compiler backend and aims to detect certain classes of bugs without any manual annotations. It first instruments the code by inserting assertions at potentially buggy locations, then generates reachability conditions for each bug and invokes Z3 to check which ones are reachable.

In the next section, we describe the particular instrumentation performed by `HackP4`; this allows us to continue analysis past the point where a bug is triggered, laying the groundwork for modelling concrete target behaviors (§IV-B). In §IV-C we present our algorithm for obtaining concrete packet bytestrings from reachability formulas. Finally, §IV-D describes the motivation for the enumerative packet generator, as well as the implementation details.

A. Checking for vulnerabilities

For each possibly-buggy line of code, we add a preceding validity check; for example, if the instruction involves reading from a header field `h.f`, we add before it a check for `!h.isValid()`. On the `then` branch, we set a location tracking variable to a unique location ID.

For each `control` and `parser` block, we declare a new metadata field to store the location ID of a bug. The special value -1 is reserved to mean that no bug was encountered. At the beginning of the parser/control block, its corresponding location variable is initialized to this value.

The following instruction in the ingress control block:

```
hdr.eth.etherType = hdr.fph.etherType;
```

is instrumented as:

```
if (!(hdr.fph.isValid() && hdr.eth.isValid()))
        meta.ingress_track = 0x79e;
hdr.eth.etherType = hdr.fph.etherType;
```

This type of conditional location tracking is added not just for assignments, but also before:

- `if` statements with possible bugs in their condition
- `apply` statements for tables whose keys contain header fields, array accesses, slices, or any other expression whose reading could be a bug
- header stack or array access to check for out-of-bounds indices
- invocations of extern functions which take possibly invalid/uninitialized fields as arguments

To detect out-of-bounds accesses, we determine the size of the indexed variable and add a guard to check if the index is lower. For example, given a Counter `bd_stats` with a declared size of 1024, the instruction:

```
bd_stats.count(meta.bd_stats_idx);
```

is instrumented as:

```
if (meta.bd_stats_idx >= 1024)
    meta.ingress_track = 0x84;
bd_stats.count(meta.bd_stats_idx);
```

Because we are strictly interested in packets that trigger a bug and are also forwarded by the switch, we need a mechanism to filter out dropped packets. For the ingress control block, we create a flag that shows whether a packet was passed or not to the traffic manager. At the beginning of ingress, we set `meta.pass_ingress = false`. At the very end, we set the flag only if the packet was not dropped. For example, on the `v1model` architecture, the check is:

```
if (standard_metadata.egress_spec != DROP)
    meta.pass_ingress = true;
```

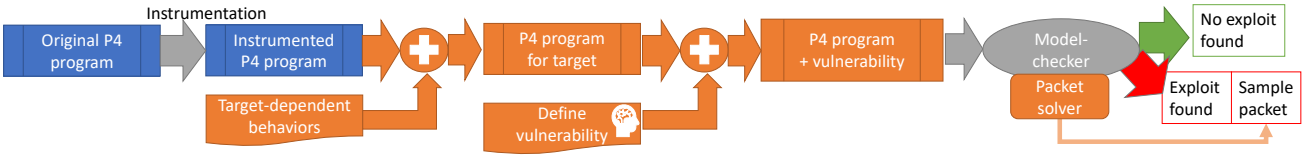

Fig. 2: Steps taken by `HackP4`. In orange, differences from `bf4`

At the end of the egress control block, we are interested only in buggy packets that will be emitted; we filter out those that were dropped during ingress processing. We also need to filter out the packets dropped during egress processing. Finally, we focus on those that triggered a bug in any block:

```
if ((meta.pass_ingress == true &&
     standard_metadata.egress_spec != DROP) &&
    (meta.ingress_track != -1 ||
     meta.egress_track != -1  ||
     meta.parser_track != -1))
    bug();
```

We then use bounded model checking to check for reachability of the bug instruction. This will decide whether there are any vulnerabilities possible in the program.

Note that we can easily enhance this condition to focus on bugs that are only triggered by certain types of packets. For example, adding `&& hdr.tcp.isValid()` to the condition, will only find buggy packets that have a TCP header.

This kind of modification must be manually performed in an intermediate instrumentation stage, but could easily be automated: `HackP4` could read a specification file describing packet constraints and translate them into such a condition.

B. Modelling concrete target behaviors

Undefined reads and writes on `v1model`. On the `v1model` architecture, in almost all contexts, reading from an invalid header field would yield the last value that a valid header contained for that field at the end of the control block. The only exception is reading from an invalid header as part of a key lookup in a table, which yields the value zero.

We overapproximate this by allowing for *any possible value* to be the result of an undefined read. By "possible", we mean one that fits the correct type of the invalid expression and is otherwise completely unconstrained.

Undefined reads and writes on TNA. Due to limited space for storing headers, the Tofino compiler can map multiple different headers onto the same memory region, as long as it can infer that these headers are mutually exclusive in the parser (i.e. there is no possible packet for which more than one such header is valid), like in Figure 1.

We model this behavior by mapping all header fields onto a single flat virtual memory space and replacing all header field access with one to this virtual memory.

During compilation, the Tofino compiler backend outputs a mapping of header fields to parsed-header values (PHVs), a set of hardware registers which are used to hold parsed packet data; the relevant information for us is:

- name of the header field
- slice of header field (represented by a most-significant-bit field and a least-significant-bit field)
- PHV index
- slice of PHV (represented by a most-significant-bit field and a least-significant-bit field)

We then map all the PHVs onto a single flat memory space, concatenating them in increasing order of their index. During instrumentation, we add a large bitfield (big enough to contain all possible PHVs) to the ingress metadata: `"bit<MEM_SIZE> mem;"`.

We modify the ingress parser such that, after every header extraction, we copy the header fields into the corresponding flat memory region, based on the field-to-PHV mapping of the compiler and our mapping of PHVs onto the flat memory:

```
packet.extract<ipv6_t>(hdr.ipv6);
...
meta.mem[1763:1756] = hdr.ipv6.hopLimit;
meta.mem[831:800] = (hdr.ipv6.srcAddr &
                     0xffffffff);
meta.mem[3331:3300] = ((hdr.ipv6.srcAddr &
   0xffffffff00000000000000000000000) >> 96);
...
```

The mapping itself is arbitrary; PHV size varies, so we just consider a bitwidth larger than the largest PHV (`maxwidth`) and map the PHV with index `x` at offset `x * maxwidth`.

In the parser and ingress, we replace each header field rvalue occurrence with the corresponding slice (or a combination of slices) from the flat memory. E.g. the table key element "`ethernet.srcAddr : ternary;`" becomes:

```
meta.mem[81:50] << 16 & meta.mem[31:16]: ternary;
```

Because the result of a bitwise `&` or << cannot be an lvalue, we replace all assignments whose lvalue is a header field with a block of assignments: the first one copies the right hand side of the assignment into a new auxiliary metadata field, which is then properly split into the memory slices associated with the field. This is also true for fields which were passed as `out` or `inout` parameters to any action or external object. Of course, we need to declare all these auxiliary variables as metadata fields.

As an example, the following assignment:

```
hdr.ipv4.diffserv = hdr.inner_ipv4.diffserv;
```

is instrumented as:

```
meta.mem_lhs_8 = meta.mem_25623_25616;
meta.mem_203_200 = meta.mem_lhs_8 & 0xF;
meta.mem_207_204 =(meta.mem_lhs_8 & 0xF0)>>0x4;
```

(In this case, `hdr.ipv4.diffserv` is mapped on the slice `[207:200]` and `hdr.ipv4.diffserv` is mapped on the slice `[25623:25616]` of the flat memory).

In order to maintain inter-block program semantics, at the very end of a control block we restore the header field values, by copying into them their corresponding memory slices.

```
...
hdr.erspan_t3_header.vlan = meta.mem[315:304];
hdr.ethernet.dstAddr = meta.mem[463:432] << 16
                     & meta.mem[5715:5700];
hdr.ethernet.srcAddr = meta.mem[431:400] << 16
                     & meta.mem[5731:5716];
hdr.ethernet.etherType = meta.mem[511:496];
...
```

After this step, there are no header field references left (except after the `extract` instruction in a parser and at the very end of a control block).

For the case of invalid reads from areas that are not overlapped with some other valid field, the value yielded seems to be consistently zero. We model this by zeroing out the entire `mem` metadata field, right before the parser.

Slice Busting Unfortunately, `bf4` cannot handle slices in the analysis step, so we add an extra instrumentation step to perform *slice busting*. Here, we aim to replace flat memory slices with variables, while preserving aliasing semantics. We cannot simply replace `mem[100:0]` with a new variable `mem_100_0`, because a more specific slice (e.g. `mem[48:32]`) might be used somewhere else and we would lose aliasing information.

We thus need to look at all `mem` slices in the code and compute the set of *atomic intervals* (i.e. those which are never split into more specific components); for each atomic interval `(h, l)`, we declare a new metadata field `mem_h_l`. We then replace each slice in the program with a bitwise AND of its atomic components, properly shifted.

Thus `x = meta.mem[12:6];` is changed into:

```
x = meta.mem_9_6 & (meta.mem_12_10 << 4);
```

After this step, there are no slices left in the program.

Header overlap exploit To illustrate the kind of vulnerabilities that require modelling header overlap, we sketch an IPv4/IPv6 forwarding program which exhibits bugs that can be employed to work around an ACL check. In the ingress control block, each packet first goes through an L3 ACL which works on either IPv6 packets or IPv4 packets; it is configured to drop all packets originating from the IPv6 address X.

```
table acl {
 key={
   ipv4.isValid(): exact; ipv4.srcAddr: lpm;
   ipv6.isValid(): exact; ipv6.srcAddr: lpm;
 } actions = { drop(); allow(); }
}
```

The packet then goes through a table that should perform the necessary rewrites – most importantly, decreasing the TTL for IPv4 packets.

```
action rewrite_ipv4() {
  ...
  ipv4.ttl--; }

table rewrite {
  key = { ipv4.isValid(): exact;
          ipv6.isValid(): exact; }
  actions = { rewrite_ipv4(); rewrite_ipv6();}}
```

At the end of the `egress` control block, we check if it is possible for an IPv6 packet from address X to be emitted by this dataplane, by manually modifying the check presented in §IV-A, abbreviated here by the expression in angle brackets:

```
if (<buggy packet was not dropped>
  && hdr.ipv6.isValid() && hdr.ipv6.src == X)
    bug();
```

We compiled this program such that the IPv4 TTL overlaps the eighth byte of the IPv6 source address (because the IPv4 and IPv6 headers are mutually exclusive). Knowing this memory layout, an attacker can craft a packet with the blocked address as source, but with the eighth byte incremented by 1. The address will not match on the ACL table, so the packet will go on undropped, reaching the rewrite table which, by decrementing the TTL field of the invalid IPv4 header, will decrement the eighth byte of the IPv6 source address, later emitting a packet from that blocked source.

The vulnerability we introduce is a controller misconfiguration: the `rewrite` table is set to do its IPv4 rewriting when the IPv6 header is *valid* and the IPv4 header *invalid*. `HackP4` then successfully generates an IPv6 packet originating from the blocked address with the eighth byte altered, as well as the control plane misconfiguration required for triggering the necessary rewrite.

Even though this example seems contrived, triggering this kind of bug is plausible in practice. Consider a 6to4 tunneling setup where IPv6 packets from an internal port are prepended an IPv4 header and forwarded externally, whereas IPv4 packets from an external port have the IPv4 header popped and are sent out on the internal port. The parser specification could identify the IP headers as mutually exclusive and map them to the same physical memory location. Whenever an IPv6 packet comes in from an internal port, rewriting inside the IPv4 packet produces side-effects into the original IPv6 packet.

Default forwarding decisions. On the `v1model`, packets are forwarded from the ingress by setting the `egress_spec` value from the `standard_metadata` field to a specific port; a special value, 511, is used to mark packets for dropping.

The `egress_spec` field is initialized by default with the value zero. If no assignment is done during ingress processing, the packet will be forwarded out of port 0.

This is easy to model for our solver, by making the initialization explicit. However, we also want to treat the lack of an explicit assignment as a bug, because default forwarding is very likely not an expected behavior.

We introduce to the ingress block a new flag `track_egress_spec`, initialized to `false`. Whenever the `egress_spec` is set in code, the flag is set. At the end of

the ingress control block, if the flag is unset, we explicitly assign 0 to `egress_spec` and mark a new bug location.

On TNA, packets are dropped by default, if no explicit forwarding decision takes place.

C. Reachability with packet processing

As our goal is to automatically generate exploits for network devices, an important capability of `HackP4` is to reason about and produce packets which can be later input into a real target in order to trigger the discovered vulnerabilities.

Ignoring packet-level constraints may lead to false positives; dealing with them is difficult because off-the-shelf SMT solvers have limited support for variable-width bit vectors. Workarounds that model packets using the theory of sequences with boolean elements or large bit vectors don't scale.

To address this, we have implemented an efficient packet reasoning and generation algorithm which enriches and leverages the capabilities of Z3 [19] to produce a variable-length packet. Our algorithm is faster than using a large bit-vector representation or using the theory of sequences as implemented in Z3. Our procedure rewrites all P4 packet processing primitives into a set of basic instructions, then solves the resulting packet equations.

Rewriting packet processing primitives. P4 architectures define a set of packet processing primitives which are used to enable parser and deparser programmability.

We convert these packet processing primitives into a set of basic instructions, that don't exhibit side effects and explicitly return their results. Table I lists these instructions and explains their meaning. Here, $\mathbb{P}$ represents the set of packets, while $\mathbb{B}^n$ the set of bitvectors of size n.

We start by instrumenting all deparsers to follow their intended append logic. Then, we replace all `extract` and `emit` instructions which operate on headers with a list of fixed size $extract_n$ and $emit_n$, one for each header field. The last step is replacing all $emit_n$ instructions as follows: $emit_n(p, y) \rightarrow concat(reverse_n(y), p)$. The resulting P4 program only contains $extract_n$, $reverse_n$, $concat$ and $empty$ packet-processing instructions.

$extract_n : \mathbb{P} \rightarrow \mathbb{P} \times \mathbb{B}^n$	extract n bits from a packet and shifts input packet right
$reverse_n : \mathbb{B}^n \rightarrow \mathbb{P}$	reverse the bit vector and convert it into a packet
$concat : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$	concatenate the two input packets
$empty :\rightarrow \mathbb{P}$	return a zero-length packet
$emit_n : \mathbb{P} \times \mathbb{B}^n \rightarrow \mathbb{P}$	append header to input packet

TABLE I: Basic packet processing primitives

Checking bug reachability. Our decision procedure starts from the bug reachability formula computed by `bf4` and feeds it to Z3. If the solver says that no bug is reachable, we deem the program correct and the procedure ends; otherwise, we have a candidate bug.

The SMT solver is capable of producing a model - an example of variable bindings which makes the bug reachable. The model corresponds to an instruction trace through the program that reaches the bug. Of these instructions, we are only interested in packet processing ones.

All packet constraints are equations that need to be satisfied for the packet to go along that particular program path. We pass these equations to our solving procedure; it produces a set of supplementary constraints to ensure packet equations are satisfied. These constraints are ranged over packet header fields which are fixed width bitvectors and thus supported by Z3. If the set of constraints is empty, this means that the current model completely satisfies the packet constraints and the procedure returns *Satisfiable*.

If all constraints are *Satisfiable*, then the resulting model satisfies packet primitives along the way. We then generate a binding of packet variables to bit strings as described by the model and return *Satisfiable*.

Solving packet equations. All packet processing constraints are equations between packet terms, that can be solved by a unification procedure similar to that described in [20]. To simplify the procedure, we observe that all packet processing equations are of the form $x = y$, where x is a packet variable and y is either a variable, the constant $empty()$, or a term of the form $concat(z, t)$ (with either z or t a variable) and that there are no cyclic dependencies between variables.

Once the packet equation solving procedure ends, if the result is *Satisfiable*, Z3 will produce a model which binds variables to concrete values. We use the results produced to generate concrete packets for targets running P4 and examine the outcomes.

D. Guessing the missing packet bits

Some bugs can only be triggered if a particular table action is applied. While the static verifier cannot know the concrete table entries, it may determine that there is some entry which, if matched, invokes an action whose body would trigger a bug. To trigger such a bug, we need the rule to be present in the table at runtime and we need to guess the concrete fields involved in the `match` part the rule.

The static component of `HackP4` can output a "packet template": the bits of a packet, together with some holes (in our example, it would output a non-TCP IPv4 packet with the destination address as a hole); the packet enumerator bruteforces all possible values in those holes, trying to stumble over a problematic packet. The performance of this can be greatly improved by domain-specific knowledge.

The packet enumerator takes two inputs:

1) a bytestring of a packet model generated by `HackP4` as described in the previous section; all relevant fields of the packet are set to the values needed in order to trigger the bug, while the other fields have arbitrary values.
2) a specification file which contains a list of bitslices to enumerate

The result is a simple "packet generator" with no knowledge of header layout and/or structure. It starts with a template (generated by bf4), then backtracks over all the bitslices specified, emitting a packet for each value.

The specification file is needed because HackP4 is not able to automatically deduce the minimal set of fields which need to be fuzzed. We list this as a current limitation in our quest for automatic exploit generation and leave it as future research.

V. Evaluation

We ran HackP4 on a number of P4 programs, such as the tutorials, the switch.p4 [21] for BMv2 and its Tofino variant, as well as the programs listed in the bf4 article [7].

We evaluated HackP4 on a machine with a 2.2GHz AMD Ryzen 5 CPU and 32GB of memory. The most complex program analyzed was the BMv2 version of switch.p4; HackP4 finished its run in around four and a half minutes, using 6GB of memory. For most other (much smaller) programs, the time and memory consumption are negligible.

We present a few exploits on real programs that showcase HackP4's ability to statically find bugs based on some specification, generate a concrete packet to trigger the bug and use packet enumeration to deal with nondeterminism stemming from table entries. We also illustrate using HackP4 to get correctness guarantees.

A. Attacking a simple NAT

Our first attacks focus on the "simple NAT"[1] depicted in Figure 3.

§4.2 of [10] describes three attacks against this program, manually discovered and investigated. We now show how HackP4 is able to detect the relevant vulnerabilities and generate concrete packets needed for two of these attacks[2].

NAT overview Packets first match an if_info table which sets metadata that tracks the packet's ingress interface and can take a drop decision. Next, the nat table is applied, which decides to forward the packet in case of a hit, or send it to the controller otherwise. Packets that must be forwarded and have positive TTL then match the lpm and forward.

To assess potential security problems, we also built a simple controller application which adds NAT table entries to previously unknown connections and re-sends the packet back into the processing pipeline. Our attacks target the *simple_switch* architecture implemented in BMv2.

Attack setup. The scenario in Figure 4, shows four targets connected to the switch: two normal targets, Alice and Bob, are connected to an internal and external port respectively, the controller program connected via the CPU port and the attacker Trudy connected to a port in the internal network. We assume that Trudy is connected to a known troublesome port and thus, an entry is included in the if_info table to drop any packet coming from this port.

Bypassing ACLs with revived packets. Given the fact that port 2 maps to the *drop* action in table if_info, it raises the expectation that all packets coming in from port 2 are dropped. We use HackP4 to check whether it is possible for such packets to bypass the port filter.

(1) We start by asking HackP4 for examples where the input port is 2, action *drop* is chosen from table if_info and constrain the packet to be output on a regular port. We achieve this by adding the following snippet to the end of the ingress block:

```
if (standard_metadata.ingress_port == 2 &&
    if_info.action_run == drop &&
    standard_metadata.egress_spec != DROP_PORT)
   bug();
```

(2) HackP4 answers *Satisfiable* and outputs a sample packet and an instruction trace. Clearly, the trace entails hitting an internal to external NAT mapping.

(3) HackP4 is analyzing only the dataplane, so it cannot know what table entries exist at runtime; it only tells us that the exploit works if there exists an internal-to-external mapping and the packet is matched on it. However, bruteforcing a NAT 5-tuple is not feasible; the search-space is 96 bits wide (two IPv4 addresses, two TCP ports). This is where our selective fuzzing strategy described in section IV-D comes into play.

We use additional information about the network to reduce the search-space. We assume the attacker knows the internal network's /24 address and may guess a likely pair of destination address and port (e.g. HTTP connections to a popular search engine). Thus, the attacker only needs to guess the last byte of the source address and the TCP source port.

We have tested this scenario on a single machine. We populate the NAT table with a randomly generated entry and start HackP4's packet enumerator, using the template resulting from static analysis and a manually defined list of fields to bruteforce.

The BMv2 simple_switch target was able to process packets at a rate of around 46 Kpps, completing the attack in an average of 357 seconds (with a packet hitting the entry within 139 seconds) over five runs.

Privilege escalation. In addition to Ethernet and IPv4, a packet can have a CPU header, designed for communication between the CPU and the P4 dataplane. This header sits above the Ethernet header and is identified by a preamble of 64 zero bits. Whenever simple_nat receives such a packet, it assumes it came from the CPU and uses the information inside the header to overwrite metadata such as the ingress port. Armed with this information, the attacker may forge a packet with this header to completely bypass all ACLs by simply pretending to have come from a different port.

We use HackP4 to check for this situation and instrument in a similar manner as for the previous bug. This time, we constrain the verifier to only look for packets whose original input port is different from that used throughout the program:

```
if ((bit<8>)standard_metadata.ingress_port
    != meta._meta_if_index10)
   bug();
```

[1] https://github.com/p4lang/p4c/blob/main/testdata/p4_14_samples/simple_nat.p4

[2] The Denial-of-Service attack described in [10] relies heavily on issues with the controller, so we ignore it.

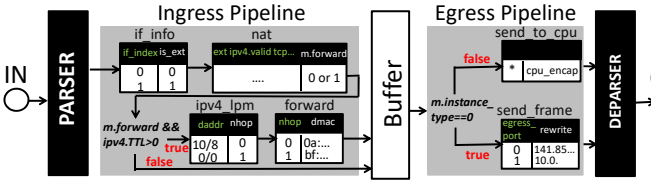

Fig. 3: P4 program example: simple NAT

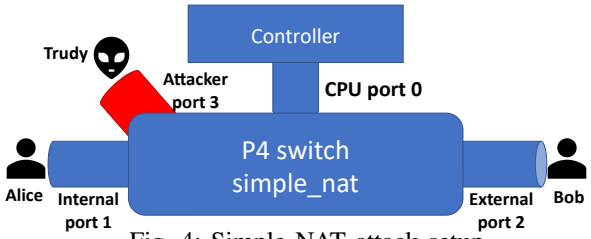

Fig. 4: Simple NAT attack setup

B. Attacking `switch.p4` on BMv2

Among other standard forwarding functionality, `switch.p4` implements ACLs – a key security function enforced in the dataplane. Adding an ACL rule requires significant control-plane processing before getting mapped into a set of P4 table entries. Is there a way to trick the dataplane into bypassing such an ACL?

We set up a simple source IPv4 ACL rule which blocks traffic from address `192.168.1.1`. We try to break this ACL using IPv4 packets sourced at this particular address. We use the following query at the end of the ingress pipeline:

```
if (fabric_present == false
  && standard_metadata.egress_spec != DROP_PORT
  && hdr.ipv4.isValid()
  && hdr.ipv4.version == 4 && hdr.ipv4.ttl >= 1
  && hdr.ipv4.srcAddr == 0xc0a80101)
    bug();
```

The checks on the IPv4 version and TTL are there to ensure that the generated packet is "valid" according to the rules installed in other tables; otherwise the packet is considered to have a malformed IPv4 header and processing would not reach this table.

First, we check whether we can break the ACL when no undefined behavior is present and then we look for an undefined behavior to trigger this policy violation. `HackP4` provides no witness in either of the situations depicted above. We can thus conclude that `switch.p4` is resilient to ACL bypass attacks even in the presence of undefined behaviors.

While this finding is expected for a well-crafted program such as `switch.p4`, it does show the value of `HackP4` in offering strong dataplane security guarantees. We believe that using `HackP4` continuously against the running P4 dataplane as a means to assess critical features (such as ACLs) is an appropriate deployment model easily amenable to automation.

Privilege escalation. In the previous example, the query at the end of ingress constrains the `fabric_present` boolean to be false. As shown in §4.3 of [10], it is possible to use the fabric header to bypass the ACL, in a way similar to the vulnerability of `simple_nat`.

The packet produced by `HackP4` has the following headers: `Ethernet`, `fabric_header`, `IPv4` with source address `192.168.1.1`. The `fabric_header` is designed to ensure communication between the CPU and the P4 dataplane. Forging such a header with the proper bypass flags, we manage to effectively bypass any ACL set up in `switch.p4` and have the switch flood packets to the outside.

VI. Related work

The problem of Automatic Exploit Generation has been successfully addressed in multiple works concerning x86 software [11]–[15]. Due in part to the difference in scope, `HackP4` does not employ any of the novel techniques presented in these works, but it is similar in principle.

There are very few works which look at the security of programmable dataplanes. Agape et al. provide a high-level overview of their security landscape [22], drawing parallels to security in software-defined networks. Our work is complementary as it provides a detailed analysis of the risks introduced strictly by the P4 dataplane and ways in which they may be exploited.

Ang Chen et al. [23] 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 behavior. This class of DoS attacks is also complementary to our work as it does not use any bugs in the P4 dataplane.

`HackP4` is, at its core, inspired by P4 verification tools [3]–[7], but goes further than state of the art in verification by moving into the uncharted territory of automatic exploit generation tailored for P4 programs.

Shukla et al. [9] describe a reinforcement-learning based fuzzer which exercises P4 bugs. `HackP4` uses a simpler template-based approach to packet enumeration based on the output of a static analysis tool.

In a previous paper [10], we ran experiments on three targets in order to document the actual manifestation of undefined behavior. This forms the basis for the modelling of target behavior described in §III. We also presented bugs on simple_nat.p4 and switch.p4, which were discovered and analyzed manually; they serve as a baseline for the kind of bugs `HackP4` should be able to detect and exploit automatically.

VII. Conclusions

In this paper, we make the case for AEG as a means of assessing the security of programmable dataplanes. We present the design and implementation of `HackP4`, a tool that finds certain classes of bugs in P4 programs (or guarantees their absence) and helps distinguish between benign bugs and vulnerabilities by generating concrete attacks if possible. Modelling concrete target behaviors enables `HackP4` to examine the processing of a packet as a whole; this allows for manual intervention to define application-specific safety properties.

Limitations. Our initial goals for `HackP4` were to automatically generate exploits starting from known undefined behaviors. However, we face limitations which prevent it from supporting all types of exploits automatically. At this point, the following steps still require human input:

- describing vulnerabilities
- deciding which packet fields need fuzzing

The former is quite tricky to automate in the absence of explicit specification. Most of the time, though, instrumenting for vulnerabilities may be achieved by testing compliance to higher level goals – e.g. in the ACL example in §V-B, a high-level spec of the ACL would have sufficed for checking ACL security compliance. These high-level goals could then be automatically translated into P4 assertions or directly into SMT formulas or axioms.

Understanding what packet fields need to be "guessed" by `HackP4`'s fuzzer starting from the P4 program requires a more involved analysis to reason about unconstrained header fields [24] resulting from the reachability procedure.

`HackP4` also inherits some limitations from its underlying verification tool, `bf4`, in terms of the types of bugs and vulnerabilities it can detect. Our evaluation of `HackP4` used somewhat simplified program scenarios to clearly demonstrate its exploit-generation capabilities in realistic conditions.

Vulnerability mitigation. Based on our comparison of existing targets' concrete behaviors, as well as observations on common habits and mistakes in P4 programs, we propose several measures to reduce exploitability risks.

For **P4 programmers** the goal is to discover and eliminate code that exhibits underspecified behavior, or *surprising behavior* (behavior that, even though well-defined, is implicit or otherwise unintuitive). We believe a tool such as `HackP4` serves as a useful step in the deployment pipeline, to help detect and assess the gravity of such issues.

Secondly, **target manufacturers and architecture designers** can harden their platforms to reduce the harmful effects of buggy code. Architecture design should avoid overloading the meaning of the same metadata field - as is the case for the `egress_spec` metadata field in the `v1model` architecture. This usually results in confusion and sometimes in exploitable vulnerabilities - such as *reviving dead packets*. Another important design decision is the default forwarding action. A packet for which no explicit forward decision is taken should be dropped by the switch.

We may distinguish a third entity involved in the deployment and functioning of a P4 dataplane, namely the **administrator**. In regards to the implicit forwarding bug, an administrator could diminish its harmful effects by leaving the "default" port unconnected, or making it the CPU port.

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, July 2014.

[2] Broadcom, "NPL: Open, High-Level language for developing feature-rich solutions for programmable networking platforms," 2019.

[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*, SIGCOMM '18, (New York, NY, USA), pp. 518–532, ACM, 2018.

[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*, SOSR '18, (New York, NY, USA), pp. 5:1–5:7, ACM, 2018.

[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*, CoNEXT '18, (New York, NY, USA), pp. 73–85, ACM, 2018.

[7] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "bf4: towards bug-free p4 programs," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 571–585, 2020.

[8] 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.

[9] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proceedings of the 2019 Workshop on Network Meets AI and ML*, NetAI'19, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2019.

[10] M. V. Dumitru, D. Dumitrescu, and C. Raiciu, "Can we exploit buggy p4 programs?," in *Proceedings of the Symposium on SDN Research*, SOSR '20, (New York, NY, USA), p. 62–68, Association for Computing Machinery, 2020.

[11] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 143–157, IEEE, 2008.

[12] S. Heelan, *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[13] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[14] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, pp. 78–87, IEEE, 2012.

[15] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, IEEE, 2012.

[16] The P4 Language Consortium, *P4_16 Language Specification*, 5 2023. v1.2.4.

[17] P4 language consortium, "Designing your own switch target with bmv2," 2019.

[18] "Intel® tofino™ series." https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[19] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. TACAS'08*.

[20] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters, "An efficient smt solver for string constraints," *Formal Methods in System Design*, vol. 48, pp. 206–234, 2016.

[21] "Switch P4." https://github.com/p4lang/switch.

[22] 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.

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

[24] M. Jonáš and J. Strejček, "On simplification of formulas with unconstrained variables and quantifiers," pp. 364–379, 08 2017.