Investigating the Vulnerability of Programmable Data Planes to Static Analysis-Guided Attacks

Conor Black
Centre for Secure Information Technologies
Queen's University Belfast
Belfast, N. Ireland
cblack39@qub.ac.uk

Sandra Scott-Hayward
Centre for Secure Information Technologies
Queen's University Belfast
Belfast, N. Ireland
s.scott-hayward@qub.ac.uk

***Abstract*—Programmable network data planes are paving the way for networking innovations, with the ability to perform complex, stateful tasks defined in high-level languages such as P4. The enhanced capabilities of programmable data plane devices has made verification of their runtime behaviour, using established methods such as probe packets, impossible to scale beyond probabilistic detection. This has created a potential opportunity for an attacker, with access to a compromised device, to subtly alter its forwarding program to mishandle only a small subset of packets, evading probabilistic detection. In practice, such subtle binary instrumentation attacks require extensive knowledge of the forwarding program, yet it is unclear whether a static analysis of compiled P4 programs to obtain this knowledge can be fast and accurate enough for an on-device attack scenario. In this work, we investigate this possibility by implementing a static analysis of P4 programs compiled to BPF bytecode. This analysis gathers sufficient information for the attacker to identify appropriate (reliably correct) edits to the program. We found that, due to predictable compiler behaviours, our analysis remains accurate even when several program behaviours are abstracted away. Our evaluation of the analysis requirements shows that, from a defensive perspective, there is scope for selectively manipulating those instructions in P4-BPF programs that are critical to attack-focused analysis in order to increase its difficulty, without increasing the number of program instructions.**

***Index Terms*—P4, static analysis, data plane security, BPF**

I. Introduction

Society's reliance on computer networks continues to grow rapidly, with the number of interconnected devices projected to hit 29.3 billion by 2023 [10]. In response, some network operators have shifted from proprietary hardware and software to open-source or programmable offerings, in a bid to tailor resources to their needs. Google [36] have leveraged merchant switching silicon to simplify network upgrades, while Microsoft have reduced switch downtime after migrating to the open-source SONiC network operating system (OS) [37].

These trends are exemplified by the growing interest in data plane programmability, spurred on by new generations of programmable application-specific integrated circuits (ASICs) [17]. While programmability has greatly expanded data plane capabilities, subtle errors in program logic can lead to difficult-to-detect bugs being installed on data plane devices. Although bugs in P4 source code may be detected by static analysis tools, such as [21], these techniques cannot detect misconfigurations that occur at runtime. Such runtime errors have been attributed to compiler bugs [32], hardware faults [8] and crucially, compromise of data plane devices [29], [39], a key target for advanced persistent threat (APT) actors [24].

State-of-the-art solutions to detecting compromised data plane devices [4] have trended towards monitoring how the devices forward crafted sets of probe packets, raising an alert if this differs from the intended behaviour. Of course, testing the forwarding behaviour of every possible packet is infeasible, particularly in stateful networks, where naive testing can take 20 hours [13]. This forces state-of-the-art packet probing solutions (e.g. P4Tester [47]) to use static analysis of P4 programs to generate a minimal set of probes that are guaranteed to test all possible program paths. While these techniques will detect alterations to existing forwarding rules, their use in detecting the addition of new forwarding behaviour is minimal, as tools cannot craft packets to trigger unknown program paths. In the new generation of programmable switches, where arbitrary packet fields can control forwarding decisions, an attacker-inserted program path that differentially forwards packets based on an atypical field (such as a magic number in the payload [11]) is unlikely to be triggered by any probe packet.

While such subtle attacks may be virtually undetectable, no previous works have investigated the feasibility of implementing them in practice, instead assuming that device compromise guarantees attack success. Crucially, to avoid detection by monitoring tools, an attacker must install a new P4 program that correctly implements all program paths of the original program, as well as any malicious alterations. Such subtle edits of compiled code are most feasibly achieved by patching the binary, rather than creating a new program. As extensively surveyed in [41], an essential stage of the binary rewriting process is binary analysis, used to discover the program's control flow and behaviour. Such analysis may be human-assisted or fully automated, yet for data plane devices whose traffic is heavily monitored in software-defined networks (SDNs), the command and control traffic required for human-assisted analysis is detectable [40]. Therefore, automated, on-device analysis is likely preferable for APT attacks, which may aim to evade detection for several years [2].

Implementing such an analysis on a compromised device may seem impractical, due to the high complexity and execution times of general static code analysis tools [44]. However,

data plane program functionality is restricted due to stringent latency requirements and attackers are unconcerned with proving program correctness, potentially reducing analysis complexity. Given the importance of program analysis to data plane attacks, it is critical that the analysis requirements are explored to inform the development of mitigations.

With this goal in mind, we investigate the feasibility of a data plane attack that uses static analysis of compiled P4 programs to inform targeted program edits, making detection by runtime monitoring tools unlikely. To this end, our first contribution (Section III) is to outline the goals and requirements of such an analysis-guided attack. Section IV comprises our main contribution: a description of our static analysis and instrumentation tool that alters P4 programs that have been compiled to Berkeley Packet Filter (BPF) bytecode (compatible with the P4Runtime-OpenvSwitch (P4RT-OvS) software switch), causing them to forward a class of previously dropped packets. This analysis is predicated on several simplifying assumptions (also detailed in Section IV) derived from observation of compiler outputs. Finally, in Section V, we find that, despite these assumptions, our analysis is both fast and accurate when tested on real-world P4 programs. In light of these results, we identify a potential programmatic mitigation to attack-focused analyses that warrants further study.

II. Background Technologies

Next, we introduce some technologies relevant to our work.

A. P4

The P4 language [7] was originally designed as a high-level alternative to programming data plane devices in specialised assembly code. To achieve this, P4 programs consist of some combination of parsers to extract packet headers from incoming packets, match-action tables to perform actions based on the values of header fields, deparsers to serialise the processed packet for output, and extern objects - immutable functions callable from the program. In this work, P4 is used as the source language, to be compiled into BPF bytecode.

B. P4Runtime

The P4Runtime Application Programming Interface (P4RT API) [28] enables control of the P4 language elements in a P4-programmable device at runtime. As illustrated in Fig. 1, P4RT communication occurs between a client located at the SDN control plane and a server program on the device. Once the client receives a request, such as instructions to install a new P4 program (the pink arrow in Fig. 1), it is passed down through a series of API calls that transform it to a device-specific request, allowing for the same P4RT protocol to be used to control heterogeneous switches. Fig. 1 depicts this workflow for a Behavioural Model version 2 (BMv2) software switch running the Stratum OS [25], where the forwarding functionality is implemented by two shared libraries (depicted in blue), functions from which are called by the P4RT Server code. In a different switch, only these shared libraries need differ. In this work, P4RT is not used directly, but is relevant

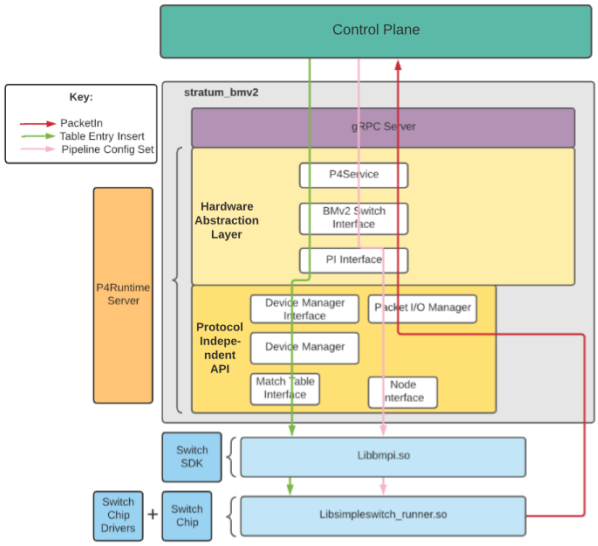

Fig. 1: P4Runtime workflow

for understanding the attack landscape. For example, previous work [5] has shown how control operations could be altered by intercepting calls to the switch shared libraries.

C. eBPF

The extended Berkeley Packet Filter (eBPF) [1] is a virtual machine (VM) that allows for safe execution of user-written code in the Linux kernel, avoiding direct kernel code alterations. eBPF programs are run by "hooking" onto predefined kernel-level events, such as packet arrivals. eBPF programs consist of 64-bit instructions, which can perform load-store operations (on both registers and stack-based memory), conditional jumps and binary arithmetic. Each eBPF program begins with a single *ctx* argument, which stores a pointer to a packet, and returns the value stored in register $r0$, which controls if a packet is forwarded or dropped. Besides the standard instructions, eBPF allows use of a limited set of *helper* functions, the most notable of which allow reads and writes to eBPF maps, a stateful memory also accessible from userspace. Given the safety guarantees required for kernel execution, eBPF programs are restricted in several ways:

- Stack memory is limited to 512 bytes.
- No floating-point numbers.
- All loops must be bounded.
- Jump offsets are compile-time constant.

These, and other safety-critical properties are enforced by the BPF verifier, which statically analyses prospective eBPF programs before they can be executed in the kernel. The verifier has been criticised for too-readily rejecting safe eBPF programs and, as a result, a number of improvements have been made, e.g., to add support for bounded loops [33].

D. uBPF & P4RT-OvS

Given that eBPF is subject to the same license restrictions as Linux kernel code, it has proven difficult to use in many projects, leading to the development of userspace BPF (uBPF). uBPF has the same instruction set as eBPF, but is located in

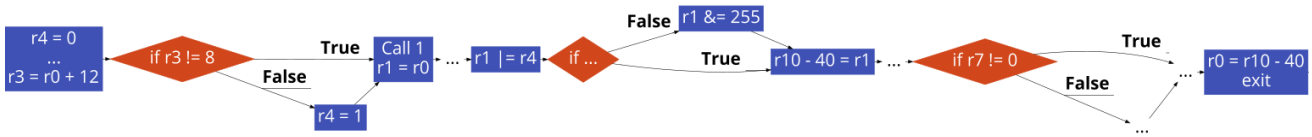

Fig. 2: Simplified version of a P4-uBPF program's control flow graph

userspace, making it easier to integrate with other projects with userspace components. One example is the P4RT-OvS switch [26]. P4RT-OvS is an extension of the Data Plane Development Kit (DPDK) version of OpenvSwitch, which uses Linux kernel bypass to maximise packet forwarding performance. This OvS-DPDK is limited by a lack of stateful packet processing and limited extensibility, which the developers of P4RT-OvS seek to remedy by integrating the uBPF VM within OvS packet processing. The authors intend for the uBPF programs for the switch to be written in P4 and compiled to BPF bytecode, using the compiler they have written. This firstly compiles the P4 code to C, before *clang* is used to compile the C program to bytecode. Loading of programs to the switch and other control operations are handled by P4RT.

For our analysis, we chose this P4RT-OvS target as it is production quality, open-source and continues to see support from the P4 community. The production suitability of P4RT-OvS is supported by a performance evaluation conducted in [26], which found that its throughput was only slightly inferior to that of OvS-DPDK with minimal packet processing and was, in fact, superior when executing some network functions. In addition, P4c-uBPF supports stateful registers, which are unsupported by the other P4-to-BPF compiler, P4c-eBPF.

A crucial requirement of our work is that the compiler outputs loaded onto the target are open-source and available for analysis. This constraint applies to the BPF bytecode output by P4c-uBPF, in contrast to the often closed-source binaries used to program ASICs or network interface cards (NICs). Determining the feasibility of analysis-guided attacks on these hardware devices is largely outside the scope of our work, particularly Section IV, which details a specific program analysis of P4c-uBPF compiler outputs. We believe, however, that the general threat model described in Section III applies to all targets and that the analysis procedure applies to all P4-BPF backends (subject to the assumptions in Section III).

III. Threat Model

In this work, we assume an attacker with the ability to run their code on a programmable switch, whose intention is to edit its P4 program to forward some class of packets that it is configured to drop. Such firewall bypass capabilities are often sought as a component of a larger attack. For example, the Cloud Snooper malware [34] utilises firewall bypass to enable command and control communication with a compromised cloud server. We do not assume a particular use of firewall bypass in our threat model, yet we do infer some attack priorities. For example, we assume that packet edits are unimportant to the attacker, since, in attacks where they may be relevant (e.g. data exfiltration), current BPF-based solutions (e.g. [38]) focus on detection or dropping of suspect packets, rather than scrubbing. Additionally, we assume that an attacker wishes to hide their modifications from the network control plane, which can monitor the compromised switch by tracking how it handles probe packets. We make no assumptions on how the switch is compromised (examples are presented in Section I), but do include attack scenarios without continuous human interaction, e.g. switch software supply chain attacks.

Control plane monitoring forces an attacker to tailor their attack program to fulfil the forwarding obligations set out by the legitimate program, changing the behaviour of only the class of packets relevant to the attack. This requirement forces the attacker to statically analyse the legitimate binary in order to craft their attack, with implications for how an attacker may successfully load the malicious program on the device. These considerations are outlined in the following subsections.

A. Binary Analysis Requirements

To ensure a stealthy attack, it is crucial that an attacker knows the criteria used by the legitimate P4 program to forward or drop packets, so their attack can accurately modify this behaviour. Determining whether a packet is forwarded or dropped in a BPF program requires resolving the value stored in the $r0$ register at program exit. As shown by the simplified BPF program control flow graph (CFG) in Fig. 2, this value can be affected by a sequence of assignment (e.g. $r0 = r10 - 40$) and arithmetic (e.g. $r1 \mid = r4$) instructions, which differ for each path taken through the program. As a result, an attacker's analysis must follow each program path individually, from exit to entry, propagating the return value expression through assignment and arithmetic instructions until it becomes a unique integer value (a technique formally known as backward slicing [43]). For example, in Fig. 2, $r0$ is assigned to $r10-40$, which is in turn assigned to $r1$, which is subject to the arithmetic expression $r1 \mid = r4$. For the program path that passes through $r4 = 1$, $r1 \mid = r4$ is trivially non-zero, meaning the path forwards packets. Unlike with bug-finding analyses of BPF bytecode, which can abstractly track variable values using bounds [23] or abstract domains [14], the attack analysis must track exact values to ensure that edits are correct. Otherwise, any resultant bugs may lead to detection.

Alongside return values, it is important to track the table, register and packet field values, which cause each program path to be taken. These criteria are specified by the conditional jumps on each program path, meaning that the constraints imposed by these instructions must be tracked by the analysis. The tracking of constraints proceeds similarly to the tracking of the return value, with the added difficulty that the final expressions often cannot be resolved to an integer like the return value. Instead, after analysis of a path, we are left with a series of expressions on BPF registers, stack variables and

pointers to function return values, which must be interpreted to discern their significance in the context of the program.

Aside from interpreting variables, another difficulty with tracking constraints is determining whether they are simultaneously satisfiable by making a single integer assignment to each variable in the constraint expressions of a single path. If not, then it is impossible for a packet to follow that path through a program. Path satisfiability problems are a major research topic in general static analysis [35], given that the complexity of solving constraints often requires the use of a satisfiability modulo theories (SMT) solver such as *z3* [12]. In the worst case, where complex mixed Boolean-arithmetic expressions define constraints, even SMT solving may be insufficient [20]. Fig. 3 illustrates the potential difficulties in determining the satisfiability of two constraints (shown in dark blue). Given the complex arithmetic involved, it is not obvious that these constraints are contradictory, requiring a single bit to be equal to both 0 and 1. While these constraints are solvable using SMT, this requires the download of a solver to the compromised device and the translation of constraints to solver API calls, which may be time-consuming and error-prone.

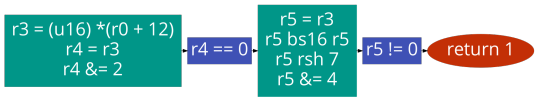

Fig. 3: Example of conflicting BPF program constraints

B. Program Injection Considerations

Assuming successful static analysis, an attacker has two options for installing their malicious program on the device: analyse and edit a running program, before unilaterally reloading it, or wait for the network operator to install a new program and analyse and edit it before installation is completed.

1) Unilateral Loading: With this approach, an attacker's program would simply retrieve the current BPF configuration from the OS, perform the analysis and editing of the program and install the edited program at a time of the attacker's choosing. The major benefit of this method is that the attacker is not time-constrained by waiting for a new program to be loaded or by time pressure to analyse and edit the program during a legitimate program update without detectable delays. However, this approach also has some major disadvantages. In the case of eBPF, installation of a new forwarding program is accomplished by invoking the *bpf() syscall*, uses of which may be monitored by a service such as *auditd* [31] to check for unauthorised program installations. This could be particularly problematic for an attacker without root privileges (e.g. within a container with only *CAP_BPF* privileges), as they may be unable to disable such *syscall* monitoring. Even where privileges are not an obstacle, replacing a BPF program unexpectedly may also break ongoing communication between the old program and the P4RT server code. Indeed, in previous work, we attempted a unilateral program change on a BMv2 software switch running the Stratum OS's P4RT Server implementation, which caused the switch to crash [5]. In addition, reloading BPF programs that use stateful maps (as most P4 programs do) can cause state loss that breaks program functionality [9], potentially leading to attack detection.

2) Timed Loading: The alternative to unilateral loading is to intercept a legitimate, controller-initiated load and edit the program before installation, to incorporate the malicious features. By utilising an expected program change, all issues with unilateral loading are avoided. The means of executing this man-in-the-middle (MitM)-style attack is not addressed here, but in a previous work we showed how it may be achieved in Stratum using a malicious shared library [5].

Despite the benefits of this method, there is a major downside for an attacker: the need to ensure that the program analysis and instrumentation run quickly enough to avoid anomalous delays to the loading of the new program, which may be detected by runtime monitoring tools. To avoid this, an attacker must ensure the program load occurs within range of the normal delay associated with program changes. For hardware switches, this problem is less acute, with fast refreshes on the Tofino switch occurring in around 50ms [3]. On BPF-based software switches, program changes take effect after just 1-6ms [22]. A MitM-style attack may still be possible on software switches, however, given that this 1-6ms timing does not include the time taken for the program change request to reach and be processed by the data plane device, or for a data plane monitoring application to detect that the new program has not yet been loaded. For example, while not directly applicable to software switch program changes, [15] tests the performance of P4 NIC and FPGA targets in updating registers and flow rules, finding delays of between 60 and 192ms, excluding control-data plane communication delay. While runtime testing tools would have to account for this delay to avoid false positives, they can themselves be the performance bottleneck. For example, for simple topologies with only a single rule change, the P4Tester runtime testing tool [47] reports detection times of around 860ms. As a result, an attacker may have several hundred milliseconds to manipulate the program without raising suspicion.

IV. Analysis Heuristics & Semantics

A. Program Behaviours

A precise analysis of general BPF programs as described in Section III-A has the potential to become very complex. However, we hypothesise that many details can be abstracted away due to predictable compiler behaviours. These behaviours, and related analysis simplifications, are described below:

1) Absence of Loops: For performance reasons, P4 programs disallow loops in the control block, but permit them in the parser (mainly to handle stacked headers such as MPLS tags), while only bounded loops are permitted in BPF programs. In theory, this combination should allow for bounded loops to be used in the P4 parser, persisting in the BPF code when compiled. In practice, however, the P4 data structure used to support the loop use case, the header stack, is not supported by the P4c-uBPF compiler. As such, our analysis can disregard loops, the handling of which is often the most difficult and time-consuming aspect of program analyses [42].

2) Individual Constraints Are Assumed Satisfiable: While we do consider scenarios where two constraints on a single variable may not be simultaneously satisfiable, we assume that any single constraint is satisfiable. In other words, we assume that every conditional jump in a program may be taken or not, excluding the potential for opaque predicates. This ensures analysis resources are not wasted on trivial satisfiability proofs.

3) Predictable Program Structure: Even after compilation from P4, the BPF bytecode retains the P4 program structure of a parser, a control block and a deparser, each of which are identifiable in the BPF bytecode. This structure allows us to assume that the return value is not altered in the deparser and that deparser conditions are irrelevant to forwarding decisions, allowing us to skip deparser blocks in our analysis.

Similarly, the implementation of P4 table reads using BPF maps takes on a predictable structure after compilation. The P4c-uBPF compiler implements each P4 table using two BPF maps: one for the main table and one to look up the default action, if the main table lookup fails. Given that the default action is a possible outcome of the main table lookup, we ignore the default table lookup. Instead, if the main table lookup fails, we transfer logical control directly to the default action, reducing the number of program paths to be tracked.

4) Arithmetic Operations: Key to analysis scalability is the ability to avoid SMT solving when uncovering infeasible program paths. This ability is predicated on an assumption of bounded complexity of arithmetic operations on any variable, v, that lacks a satisfying assignment on a given path. Specifically, as detailed by the Sat predicate in Section IV-B, we assume that any such v is eventually resolved to a single integer - either by program instruction (e.g. $v = 0$) or equality constraint (e.g. $v == 0$) - or that contradictory constraints are subject to identical arithmetic expressions.

While these conditions are found in most real-world programs, they are not strictly guaranteed. However, as observed in Section V, the presence of unhandled arithmetic on unsatisfiable paths may not impact final accuracy, due to path redundancy. Even where a subset of final paths are misclassified, alternative paths can be selected for binary instrumentation.

While not directly related to path classification, we also assume that a variable v_i is subject to at most one arithmetic operation with a variable operand, v_j. This limits expressions to the form $((v_i \ op \ int)...) \ op \ ((v_j \ op \ int)...)$, where $op \in \{+, -, \times, \div, \%, \wedge, \vee, byteswap, \ll, \gg\}$. While greater complexity was not observed, this assumption solely simplifies state storage and may be relaxed without reducing accuracy.

5) Non-Overlapping Stack Variables: We observe that stack variables are stored in non-overlapping regions and that all writes manipulate only one region at a time. This assumption introduces a crucial analysis simplification, in that it allows us to avoid tracking the bounds of every stack variable to ensure that there are no overlaps between them, which would introduce hidden constraints on the overlapping bits.

6) Pointer Values: In the programs that we have analysed, $r0$ is always used to store pointers to the first byte of function return values. Additionally, any dereferencing of $r0$ with a positive offset (e.g. $*(u64*)(r0 + 12)$) refers solely to non-overlapping packet values. These behaviours contribute greatly to analysis simplification. Firstly, as with stack variables, we don't have to consider hidden constraints between packet values due to overlapping bounds. In the case of pointers, we also avoid the complexity of tracking pointer arithmetic, which often appears in general BPF programs.

B. Implemented Analysis

We implement a program analysis that limits the complexity of tracked program behaviours, in line with the assumptions in Section IV-A, to remain accurate without requiring SMT solving. Our analysis runs backwards on BPF instructions, which have been labelled to indicate the boundaries of the basic blocks of the program's CFG and categorises each program path as 'pass' if it forwards packets, 'drop' if it drops packets and 'invalid' if the path constraints are unsatisfiable. As discussed, we simplify the CFG such that it skips over deparser blocks and blocks containing default table reads. Next, we outline the analysis state and semantics.

1) Analysis State: Given that the analysis must be path-sensitive, the analysis state Σ is actually a collection of substates σ, one for each path through the program, such that $\Sigma = \{\sigma_0, ..., \sigma_n\}$. At each branch instruction, new substates are created for each predecessor block, with each incorporating the constraint associated with the branch taken.

Each substate can be defined by the list of basic blocks that the path has traversed, a list of constraints on program variables and a Boolean to indicate the validity of the path.

$$\sigma = (IDMap, RangeMap, blocks, sat)$$

As described in subsequent sections, the composition of these constraints changes frequently as the analysis progresses, so we choose to store these constraints as a set of maps from variables and bit-ranges to positions in the constraint system. This allows for easy reconstruction of the constraints, K, by changing the variable mapped to a given position.

P_l and P_r are used to signify positions on the left hand side (LHS) and right hand side (RHS), respectively, of constraints. The LHS of constraints consist of variable identifiers and bitvectors, while the RHSs consist of bitvector ranges, an additional Boolean to indicate a $\neq$ constraint and a numerical identifier of a linked constraint (used when the original conditional is of the form $r_i \ cmp \ r_j$, where $cmp \in \{=, \neq, >, \geq, <, \leq\}$).

$$IDMap = \{regs, stack, call, pktbytes, \{0,1\}^{64}\} \rightarrow P_l$$

$$RangeMap = ranges \rightarrow P_r$$

$$ranges = ((a, b), \{t, f\}, \{\emptyset, 0, ..., n\}), a \leq b \in \{0,1\}^{64} \cup \{\cdot\}$$

Given that the RHS of each constraint is totally defined by $ranges$, it is sufficient to define P_r as the power set of natural number constraint identifiers.

$$P_r = \mathcal{P}(\{0, ..., k\})$$

The LHS of the constraints is harder to define, given that it can consist of both variables and arithmetic operations applied

to them. As well as specifying the constraints that an ID is mapped to, its position in the arithmetic expression on the LHS of each constraint must be specified along with any arithmetic operation applied to it. We define this position using the syntax tree of the expression, where each x_i in the definition below indicates a depth level of the tree, and can be set to either 0 or 1 (or $\emptyset$ if that depth is not reached) depending on if it is the LHS or RHS of the subexpression.Finally, the arithmetic operations applied to each ID are stored with the first ID on the right of the operation at the lowest level of the syntax tree.

$$P_l = \mathcal{P}(\{p_0, ..., p_n\}), p_i = (\{cmp, \emptyset\}, k, (x_0, ..., x_m)), x_i \in \{\emptyset, 0, 1\}$$

Traversal of the program instructions, P, continues on each substate individually, until one of σ_{pass}, σ_{drop} or σ_{inv} is reached. Constraint 0 for each substate concerns the program return value, with σ_0 containing the constraint $r0 \in \{0, 1\}$.

$$\sigma_{pass} = \sigma[\{1, 1\} \rightarrow (\emptyset, 0, (0, \emptyset, ..., \emptyset))), 0 \in blocks, sat = true]$$

$$\sigma_{drop} = \sigma[\{0, 0\} \rightarrow (\emptyset, 0, (0, \emptyset, ..., \emptyset))), 0 \in blocks, sat = true]$$

$$\sigma_{inv} = \sigma[sat = false]$$

2) Sat Predicate: The sat Boolean in each substate σ is set by the Sat predicate, which is applied after the execution of conditional and assignment instructions if the IDs involved did not map to $\emptyset$ before the instruction's execution. The semantics of Sat differ by instruction, and so are defined alongside the applicable instructions in subsequent subsections. If the Sat predicate returns $true$, traversal of the path continues as normal. However, if it returns $false$, sat is set to 0 for that substate and execution on that substate is halted.

3) Assignment Instructions: Given that the analysis abstracts both registers and pointers as generic 64-bit variables, we can define a simple assignment semantics as follows.

$$I[ID_i = ID_j] = \sigma \rightarrow \sigma[ID_i \rightarrow \emptyset, ID_j \rightarrow \{P_{ID_i}\} \cup \{P_{ID_j}\}]$$

Simply, the positions of the destination ID, i, are transferred to the source ID, j, and the destination ID is mapped to the empty set. This can impact path satisfiability in two ways. Firstly, if $ID_j \in \{0, 1\}^{64}$, then we can evaluate any constraint that ID_i was originally mapped to, if all other positions in the constraint are occupied by bitvectors. So, in the case where ID_j is equal to some bitvector y, $Sat_{ID_i=y}$ can be defined as follows, where K_y are the LHSs of constraints involving y:

$$Sat_{ID_i=y} = \begin{cases} true \text{ if } Eval(K_{y_i}) \in \{range_{K_{y_i}}\} \cup \{\infty\}\ \forall K_{y_i} \in K_y \\ false \text{ if } \exists\, K_{y_i} \text{ s.t. } Eval(K_{y_i}) \notin \{range_{K_{y_i}}\} \cup \{\infty\} \end{cases}$$

where $Eval$ returns the integer result of the expression if all elements are bitvectors and ∞, if not.

The second Sat test that may apply for assignment instructions takes place when both $P_{ID_i}, P_{ID_j} \neq \emptyset$ and $ID_i, ID_j \notin \{0, 1\}^{64}$. In this case, should one of the constraints in K_{ID_i} or K_{ID_j} be of the form $ID = y$, then $Sat_{ID_j=y}$ is carried out as outlined above. Another check is performed when this does not apply. In these cases, if $\exists\ K_{ID_{i_l}}, K_{ID_{j_m}}$ such that $K_{ID_{i_l}} = K_{ID_{j_m}}$, except for in position 0, then $Sat_{ID_i=ID_j}$ can be partially defined as follows, where $x \in \{0, 1\}^{64}$:

$$Sat_{ID_i=ID_j} = \begin{cases} true \text{ if } \exists x \text{ s.t. } x \in range_{K_{ID_{i_l}}} \cap range_{K_{ID_{j_m}}} \\ false \text{ if } \nexists x \text{ s.t. } x \in range_{K_{ID_{i_l}}} \cap range_{K_{ID_{j_m}}} \end{cases}$$

In other words, where there are constraints where ID_i and ID_j are both in the first position before the assignment and each constraint's arithmetic operations are identical to each other, they are only satisfied if the ranges overlap. If the constraints are not of this form, they are not checked.

4) Arithmetic Operations: As alluded to in Section IV-A4, we only track arithmetic operations of the form $r_i\ cmp\ r_j$ where $r_i, r_j \in regs$ for a constraint K_{r_i} if $\forall x_i \in P_{r_i}$, $x_i = 0$. The semantics for this case are defined below. Note that only $regs$ and bitvectors can be involved in arithmetic.

$$I[r_i\ cmp\ ID_j] = \sigma \rightarrow \sigma[r_i \rightarrow P_{r_i}[\forall(..., x_i, \emptyset, ...) \rightarrow (..., x_i, 0, ...)], ID_j \rightarrow (op, (..., x_i, 1, ...))]$$

5) Conditions: Conditional instructions prompt the creation of a new constraint, by forcing a register value to lie within the range specified by the branch taken (determined by the values in $blocks$). For each branch, there are two scenarios: where the condition is of the form $r_i\ cmp\ int$ and where it is of the form $r_i\ cmp\ r_j$. In the latter, to avoid having complex expressions at either side of the condition, we create a separate constraint for each register and link them by index. When one of the $regs$ is resolved to a bitvector, the placeholder $(\cdot)$ is replaced with that bitvector in the linked condition (not shown in the assignment semantics above). In the simpler case, where the RHS is an integer, the relevant range is assigned to the RHS and r_i is assigned to the LHS. Semantics for both instructions are shown below, with only the r_i case included for $r_i\ cmp\ r_j$ and only a single $\leq$ predicate considered, due to space constraints.

$$I[r_i \leq y] = \sigma \rightarrow \sigma[r_i \rightarrow (\emptyset, k, (0, \emptyset, ...)), ((0, y), f, \emptyset) \rightarrow k_{r_i}]$$

$$I[r_i \leq r_j] = \sigma \rightarrow \sigma[r_i \rightarrow (\emptyset, k, (0, \emptyset, ...)), ((0, \cdot), f, k_{r_j}) \rightarrow k_{r_i}]$$

Satisfiability in this case is determined exactly as in the assignment case, apart from when the instruction is of the form $r_i\ cmp\ r_j$. In this case, no satisfiability checks are performed.

6) Function Calls: The final instruction type to be handled is a call instruction, which causes all of r_0's positions to be allocated to $call_i \in call$, with no satisfiability check.

C. Path Pruning

Even assuming the accuracy of the implemented program analysis, its output can be difficult to interpret when many paths are returned. Often, however, these paths are functionally identical, as the differing conditions do not contribute to forwarding decisions. For example, a condition may control the data being written to a register after the return variable is set. Including identical paths not only hinders understanding, but overwriting these redundant conditions in our attack (described in Section IV-D) increases program size and may slow it down. As an additional component of our analysis, we seek to merge

duplicate paths, by reducing all paths to their key conditions and deleting duplicates. We define a key condition as any condition that, if its decision was reversed for any particular path that forwards packets, the path would drop packets.

We remove duplicate paths using a two-step algorithm. Firstly, we store all conditions that appear in 'drop' paths in a 'fail list', regardless of whether the conditional jump was taken in a path. Then, for every condition in each 'pass' path, if it is not in the fail list, that condition is deemed irrelevant and constraints arising from that condition are removed from the path, as any condition not present in 'drop' paths cannot be responsible for dropping packets. The second step of the algorithm involves removing functionally identical paths after the removal of irrelevant constraints. Paths are deemed identical if one path's constraints are a subset of those of another path, as the additional constraints are clearly irrelevant to forwarding decisions. The longer path is removed. The second form of path equivalence is where the constraints for two paths are identical, save for those arising from the same condition, where the jumps taken are different. In this case, since either direction leads to packet forwarding, the paths are functionally equivalent and one can be removed.

D. Binary Instrumentation

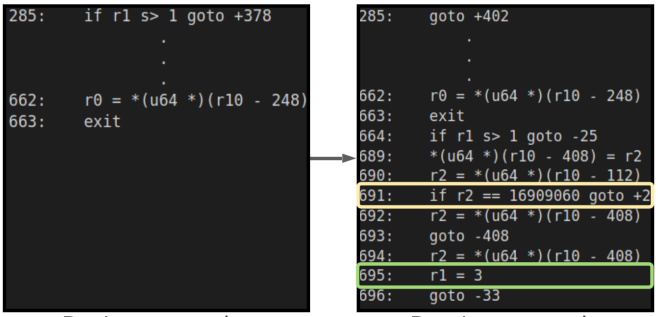

Fig. 4: Attack code instrumentation

Our algorithm for inserting attack code into binaries is simple, in that it replaces each key conditional of one chosen path with additional code (as shown in Fig. 4) that checks for a packet characteristic of interest (source IP, yellow in Fig. 4). If this is present and the value in the register does not satisfy the condition required to forward the packet, we simply change its value (green in Fig. 4) and redirect control flow according to the new value. Otherwise, we direct control flow as originally intended, so as not to impact other packets. We add our code at the end of the original code and overwrite the original conditions with *goto* instructions that jump to our appended code. This is to avoid invalidating other jump offsets.

While this attack is indeed simple for all of the programs we studied, it is enabled by several assumptions about the simplicity of the program structure. Perhaps the most consequential assumption is that the variable of interest to the attacker has been parsed and is stored on the stack, which is not guaranteed for general BPF programs. The second assumption is the presence of a constant-valued stack variable that can be used as temporary storage. However, this does hold for most programs, due to the parsing of constant header fields (e.g. ethertype).

V. Evaluation & Discussion

In evaluating our analysis and attack code, we focus on two key metrics: the accuracy of the analysis in categorising paths (as 'pass', 'drop' or 'invalid') and overall execution time. In addition to accuracy and execution time, we decided to track the instructions that altered analysis state (% Impactful Instructions) and instrumentation overhead (% Added Instructions) as an indicator of the simplicity of the technique. To determine these metrics, we had to test our work on real-world P4 firewall programs compiled with the p4c-uBPF compiler. However, given the dearth of open-source P4 programs written for a uBPF target, we modified programs written for other targets to compile with p4c-uBPF. We made several adjustments:

- Replaced unsupported hash and counter externs with uBPF-compatible hashes and registers, respectively.
- Grouped together unused packet fields in parsing, to avoid running out of stack space.
- Removed some additional program features if the program was too large to compile.

While we endeavoured to ensure that the compiled C code was kept as faithful to the original as possible, we were forced to make some adjustments to fix bugs in the implementation. These changes were: initialising *null* return values with 0s to allow the remainder of the program to run, removing references to the *pkt_len* metadata field (which was unsupported) and ensuring that the default table action was always invoked when a table lookup returned *null*.

The results of running our analysis on five open-source programs (and one edited variant) are shown in Table I. One striking result is the low execution time of the combined analysis and instrumentation, particularly for programs with few valid paths. While this indicates that our attack method could feasibly be used in the MitM-style attacks described in Section III-B2, it becomes less certain as program complexity and analysis time increase. Indeed, the relationship between execution time and the number of program paths (the 'path explosion' problem) is a key challenge for path-sensitive program analyses such as [6]. In the case of P4-uBPF, however, the absence of loops (with the current compiler) and strict latency requirements limit the number of paths in practice.

Another striking result is the low percentage of impactful instructions that alter the analysis state (under 14% in all cases). Fig. 5 illustrates this for the Simple Firewall program [27], with green instructions being impactful, yellow instructions requiring map lookups but no meaningful action and grey instructions able to be skipped entirely. Given that the Simple Firewall program is analysed with 100% accuracy, it is interesting how many instructions are irrelevant to analysis, particularly those dedicated to parsing and deparsing unused packet fields. Unsurprisingly, programs with a higher percentage of impactful instructions also yield higher execution times.

Finally, the most interesting analysis from a correctness perspective is that of [46], as it is was not 100% accurate. Of the

TABLE I: Performance - Static Analysis (Path Accuracy), Code Instrumentation (Instruction No.) & Execution Time

	# Lines	# Blocks	Total # Pass Paths	Total Path Accuracy (%)	Reduced # Pass Paths	Reduced Path Accuracy (%)	% Added Instructions	% Impactful Instructions	Execution Time (ms)
P4Knocking (w/o tables)	684	24	3	100	3	100	5.3	4.4	6
P4Knocking [45]	559	27	3	100	2	100	4.8	6.6	7
Simple Firewall [27]	679	33	35	100	7	100	2.8	11.8	37
Poise [18]	708	55	23	100	6	100	6.1	12.3	42
BloomFilterFirewall [16]	749	20	5	100	4	100	2.9	6.3	6
P4ResearchDoS [46]	757	48	55	74.5	12	100	7.7	13.9	149

Fig. 5: Simple Firewall program instructions labelled by program section and highlighted by analysis impact

55 'pass' paths found by the analysis, 14 of these were actually invalid, yielding an accuracy of just 74.5%. The conflicting conditions in the invalid paths were not detected by the Sat predicate, as their complexity is not handled in the analysis semantics. For example, one path yielded the constraints $x \wedge 65281 = 512$ and $(((x \wedge 65281)\ byteswap_{16}) \wedge 2) = 0$, where x is a 16-bit variable. In this case, the 2nd LSB of x must be both 1 and 0 simultaneously, a contradiction. While these conflicting conditions result in false positive 'pass' paths being reported, after path pruning, all inaccurate paths are deleted. While this behaviour is desirable, it is not guaranteed and only occurs due to the redundancy of the contradictory conditions. For another program, these contradictions could lead to misclassification of paths. Despite this, for all programs analysed, trivially valid paths were found for instrumentation.

A. Potential Mitigations

Given that a low number of impactful instructions is a key determinant of attack success, increasing the number of such instructions in a given program appears to be a simple attack mitigation. However, such a step is likely to lead to slower program execution times and, thus, an unacceptable reduction in throughput. Despite this, given that both Fig. 5 and the results in Table I suggest that the programs we have analysed are dominated by instructions that do not contribute to forwarding decisions, it remains an open question whether some of these may be removed and replaced by impactful instructions, leaving no overall reduction in performance. Indeed, analysis carried out by the creators of the P4c-uBPF compiler [26], suggests inefficiencies in compilation from P4, resulting in unnecessary instructions that may be removed. While the presence of such redundancies depends on continued compiler inefficiencies, their identification and replacement with analysis-sensitive instructions is a focus of future work.

VI. Related Work

While our work is unique in its narrow focus on enumerating the paths of P4-uBPF programs, several other works either share elements of our approach or centre on a similar target. Technically, worst-case execution time (WCET) analyses such as Trickle [6] are most similar to our analysis, since they analyse compiled binaries to find invalid paths. There are many key differences, however. Trickle, for example, in common with other WCET analyses, uses implicit, rather than explicit, path enumeration to avoid path explosion. In addition, Trickle analyses only the data flow of the longest path.

Our binary instrumentation process also differs significantly from other solutions (surveyed in [41]), given that these are generally designed for hardening of general purpose binaries. Technically, our instrumentation process relies on trampoline instructions, similar to PEBIL [19]. However, we avoid the challenges of patching indirect, variable-length jump instructions tackled in PEBIL due to their absence in BPF bytecode.

Network programs have also been the object of static analysis, most often to verify program correctness. P4v [21] is the most prominent such work focused on verifying P4 source code. While P4v aims to discover semantic program properties in common with our work, it differs completely in its methodology and focus on formal correctness.

The PREVAIL verifier [14] attempts to verify eBPF program safety using abstract interpretation-based static analysis to guarantee, for example, in-bounds memory accesses. As with P4v, this differs from our work in purpose, but also in precision. While our work focuses on a narrow subset of all instructions, evaluating them precisely, PREVAIL analyses complex operations such as pointer arithmetic, sacrificing the tracking of exact variable values for scalability.

Finally, Klint [30], seeks to verify generic, x86-64 network function binaries without abstracting variable values like PREVAIL. Similar to our work, Klint incorporates assumptions about program structure to allow scalable analysis. These assumptions, such as discounting memory allocations during packet processing, are, however, intended to simplify verification, contrasting with our focus on infeasible path detection.

VII. Conclusion

As data plane forwarding devices become increasingly recognised as a potentially high-value attack target, it is important that we consider how an attacker might attempt to manipulate the forwarding behaviour for their own ends in the age of programmability. In this work, we take a first step to understanding the requirements of a data plane program editing attack, by implementing a basic static analysis that nevertheless gives an accurate insight into the behaviour of BPF programs compiled from P4. Our results show that P4-uBPF programs contain a low percentage of impactful instructions, enabling low execution times for our program analysis and instrumentation. Achievement of such execution times indicates that program editing attacks can be quick enough to be imperceptible to state-of-the-art runtime monitoring tools. We hope that, as a result, future work can use this insight to implement program-level defences against data plane attacks by frustrating this type of real-time, on-device analysis.

References

[1] eBPF Documentation. https://ebpf.io/what-is-ebpf.

[2] A. Alshamrani, S. Myneni, et al. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Communications Surveys & Tutorials*, 21(2):1851–1877, 2019.

[3] A. Bas. Leveraging Stratum and Tofino Fast Refresh for Software Upgrades. In *ONF Connect*, 2018.

[4] C. Black and S. Scott-Hayward. A Survey on the Verification of Adversarial Data Planes in Software-Defined Networks. In *Proceedings of the 2021 ACM International Workshop on Software Defined Networks & Network Function Virtualization Security*, pages 3–10, 2021.

[5] C. Black and S. Scott-Hayward. Adversarial Exploitation of P4 Data Planes. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 508–514. IEEE, 2021.

[6] B. Blackham, M. Liffiton, et al. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[7] P. Bosshart, D. Daly, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Comm. Review*, 2014.

[8] P. Bressana, N. Zilberman, et al. Finding hard-to-find data plane bugs with a PTA. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020.

[9] J. Burton. BPF Map Tracing: Hot Updates of Stateful Programs. In *Linux Plumbers Conference*, 2021.

[10] Cisco. Cisco Annual Internet Report (2018–2023). https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, 2020.

[11] A. Cui, J. Kataria, et al. Killing the myth of Cisco IOS diversity. *USENIX Worshop on Offensive Technologies*, 2011.

[12] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] S. K. Fayaz, T. Yu, et al. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016.

[14] E. Gershuni, N. Amit, et al. Simple and precise static analysis of untrusted linux kernel extensions. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[15] H. Harkous, M. He, et al. Performance Study of P4 Programmable Devices: Flow Scalability and Rule Update Responsiveness. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–6. IEEE, 2021.

[16] S. Ibanez. P4 Tutorial Firewall. https://github.com/p4lang/tutorials/blob/master/exercises/firewall/solution/firewall.p4, 2019.

[17] Intel. Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2018.

[18] Q. Kang, L. Xue, et al. Programmable In-Network Security for Context-aware BYOD Policies. In *29th {USENIX} Security Symposium*, 2020.

[19] M. Laurenzano, M. Tikir, et al. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183. IEEE, 2010.

[20] B. Liu, J. Shen, et al. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *USENIX Security Symposium*, 2021.

[21] J. Liu, W. Hallahan, et al. P4v: Practical verification for programmable data planes. In *2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[22] S. Miano, M. Bertrone, et al. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[23] L. Nelson, X. Wang, et al. A proof-carrying approach to building correct and flexible BPF verifiers. In *Linux Plumbers Conference*, 2021.

[24] NSA. Chinese State-Sponsored Actors Exploit Publicly Known Vulnerabilities. Technical report, 2020.

[25] B. O'Connor, Y. Tseng, et al. Using P4 on fixed-pipeline and programmable Stratum switches. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[26] T. Osiński, H. Tarasiuk, et al. A runtime-enabled p4 extension to the open vswitch packet processing pipeline. *IEEE Transactions on Network and Service Management*, 2021.

[27] T. Osiński and M. Budiu. Simple Firewall. https://github.com/p4lang/p4c/backends/ubpf/tests/testdata/test-simple-firewall.p4, 2020.

[28] P4 Language Consortium. P4 Runtime Specification. *Website, https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html*, 2017.

[29] G. Pickett. Staying persistent in software defined networks. *Black Hat Briefings*, 2015.

[30] S. Pirelli, A. Valentukonyte, et al. Automated Verification of Network Function Binaries. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[31] J. Pope, F. Raimondo, et al. Container Escape Detection for Edge Devices. In *ACM Conf. on Embedded Networked Sensor Systems*, 2021.

[32] F. Ruffy, T. Wang, et al. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[33] M. Rybczyńska. Bounded loops in BPF for the 5.3 kernel. https://lwn.net/Articles/794934/, 2019.

[34] S. Shevchenko. Cloud Snooper Attack Bypasses AWS Security Measures. https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-cloud-snooper-report.pdf, 2020.

[35] Q. Shi, X. Xiao, et al. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[36] A. Singh, J. Ong, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 2015.

[37] R. Singh, M. Mukhtar, et al. Surviving switch failures in cloud datacenters. *ACM SIGCOMM Computer Communication Review*, 2021.

[38] J. Steadman and S. Scott-Hayward. DNSxP: Enhancing data exfiltration protection through data plane programmability. *Computer Networks*, 2021.

[39] K. Thimmaraju, B. Shastry, et al. Taking control of sdn-based cloud systems via the data plane. In *Symposium on SDN Research*, 2018.

[40] X. Wang, K. Zheng, et al. Detection of command and control in advanced persistent threat based on independent access. In *2016 IEEE International Conference on Communications (ICC)*, 2016.

[41] M. Wenzl, G. Merzdovnik, et al. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 2019.

[42] X. Xiao, S. Li, et al. Characteristic studies of loop problems for structural test generation via symbolic execution. In *28th IEEE/ACM International Conference on Automated Software Engineering*, 2013.

[43] B. Xu, J. Qian, et al. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[44] H. Xu, Z. Zhao, et al. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing*, 17(6):1243–1256, 2018.

[45] E. O. Zaballa, D. Franco, et al. P4Knocking: Offloading host-based firewall functionalities to the network. In *23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020.

[46] M. Zhang. DDoS Mitigation. https://github.com/zhangmenghao/p4research/blob/master/DoS/bmv2/p4src/syntry.p4, 2018.

[47] Y. Zhou, J. Bi, et al. P4tester: efficient runtime rule fault detection for programmable data planes. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019.