Adversarial Exploitation of P4 Data Planes

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 data planes can support flexible and feature-rich networks. However, the network operator must have confidence that the network data plane correctly implements the specified policies. To address this, data plane testing and verification mechanisms have been proposed, which, in general, trust the data plane devices to behave faithfully. A few current solutions recognise that one or more of the network devices may be under the control of a malicious adversary but do not address either the enhanced capabilities or motivations of an attacker in a modern P4-programmable data plane. Furthermore, the ability of an attacker to utilise these enhanced capabilities in an exploit has not been investigated. In this paper, we address this knowledge gap by means of a case study in which we assume the role of an attacker in an open-source implementation of a P4-programmable software switch and attempt a range of methods to exploit the program running on that switch. We find that attacks that exploit both the programmability and statefulness of the P4 switch are indeed possible, and discuss the impact of our findings with proposals for future adversarial data plane verification mechanisms to address this new threat model.**

***Index Terms*—P4, Programmable Data Planes, Data Plane Verification, Software-Defined Networks**

I. Introduction

The centralised control of network forwarding devices made possible by software-defined networking (SDN) has revolutionised the development of network applications by enabling a shift towards logically centralised applications (e.g. load balancers [36] and firewalls [16]) with network-wide visibility. The key enabler for these applications in early OpenFlow (OF)-based [24] SDNs was the central collection and processing of statistics from network devices at the control plane (CP), as these statistics could inform network-wide policy updates. However, despite the benefits, this architecture was quickly recognised as excessively restrictive for many stateful applications [5] due to the latency introduced by switch-controller communication to update or read state variables.

Now, the surge in popularity of programmable network data planes (DPs) has changed the landscape for application developers. This is, in large part, due to the Reconfigurable Match Table (RMT) architecture's offer of programmability without sacrificing performance [7] and the emergence of the P4 programming language [6] as the de facto standard for programming the DP. The support for stateful data structures and hash functions in P4-programmable switch chips, such as Barefoot's Tofino [1], and native language support for mathematical operations, such as bitwise addition on packet header fields, have allowed application developers the flexibility to offload relatively simple tasks to DP devices. This not only allows for significant reductions in latency for stateful network functions (e.g. stateful firewalls, load balancers) by allowing them to reside entirely in the DP, but can also reduce the workload on more complex analysis applications (e.g. machine learning applications) by pre-processing statistics [25].

A major consequence of making DPs both stateful and programmable is an increased difficulty in verifying that DP behaviour corresponds to high-level network policies. As complex programming constructs make the deduction of intended device behaviour more difficult, and stateful changes to DP behaviour can occur without controller direction or visibility, exemplar OF-based verification solutions such as Veriflow[1] [19] become inapplicable. To address this, significant research has been undertaken on P4 verification and testing, using techniques such as static analysis (incl. symbolic execution) of P4 programs and table entries [15], [22], [23], [32], runtime testing of P4 switches [26], [31] and analysis of runtime statistics or postcards[2] provided by the DP [20], [38], [42].

Given that many of these verification solutions are intended to catch bugs, they operate under the implicit assumption that DP devices behave honestly. Given that attacks that exploit weaknesses in SDN-ready DP switches have already been demonstrated [28], [34], there is a clear possibility that an attacker with control of a DP switch may use this trust to falsify runtime statistics, evading these verification solutions. As it stands today, there are only a few DP verification solutions that acknowledge this potential presence of a malicious actor. We refer to these as *adversarial data plane verification* (ADPV) solutions; mechanisms that attempt to detect anomalous behaviour from compromised DP switches. These solutions differ from non-adversarial verification mechanisms in that their designs implicitly assume that an attacker may attempt to hide their activities from the CP, meaning that data received from an individual switch is not automatically trusted.

State-of-the-art ADPV solutions can be largely divided into three categories: those that use data plane cryptography to

[1]Veriflow is a real-time network verification solution that monitors OpenFlow messages between the DP and CP to detect attempted changes to the network that violate high-level policies.

[2]A postcard is a packet header that includes information about which switches a packet has traversed and which forwarding rules were applied.

978-3-903176-32-4

validate packet paths [29], [40], those that use probe packets to test forwarding behaviour [9]–[11] and those that distribute statistics collection across multiple switches to account for switches that may be misbehaving [13], [17], [30], [33], [41].

While the techniques employed vary, these solutions all share an assumption of a DP in which all functionality is stateless and an attacker's capabilities are limited to logical compositions of drop and inject operations on DP packets. However, as network operators move to use the DP in novel ways as enabled by P4, it is not yet clear to what extent an attacker may do the same. The establishment of this is crucial to the development of updated ADPV mechanisms, not only because porting OF-based ADPV solutions to a P4 DP may result in blind spots in attack coverage, but also because there may be practical limits to an attacker's capabilities that could be leveraged by detection mechanisms.

We identify three main contributions of this work. Firstly, we present a unique analysis of the P4-SDN architecture from the perspective of an adversarial attacker identifying the layers of abstraction and the potential for exploitation (Section II). Secondly, by means of a case study, we showcase two possible attacks that take advantage of the statefulness and programmability of the P4-SDN architecture to alter the switch's forwarding behaviour (Sections III and IV) Finally, we propose a new approach to ADPV for stateful P4 DPs to address this new threat model (Section V).

II. P4-SDN Environment

The elements in the P4-SDN architecture (shown in Fig. 1) are introduced as a foundation for analysis of adversarial exploitation of the DP.

A. P4

The P4 language was originally designed as a high-level alternative to programming DP 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 certain header fields, deparsers to serialise the resultant packet that is to be sent to an output port and extern objects, which are essentially black box functions. The most pertinent extern to this work is the register, a stateful array that allows the programmer to store data that persists beyond the processing of a single packet. The exact number and format of each construct in a P4 program depends on the architecture for which it is designed. Fig. 1, for example, depicts the Portable Switch Architecture (PSA), which is intended to be the standard architecture to which a variety of network switch chips will conform.

B. P4Runtime

The P4Runtime (P4RT) API [27] enables control and monitoring of the P4 language elements in a PSA-compliant switch at runtime. This communication is performed by a gRPC server on each switch and a gRPC client (usually running on an SDN controller). It is considered best practice to protect this

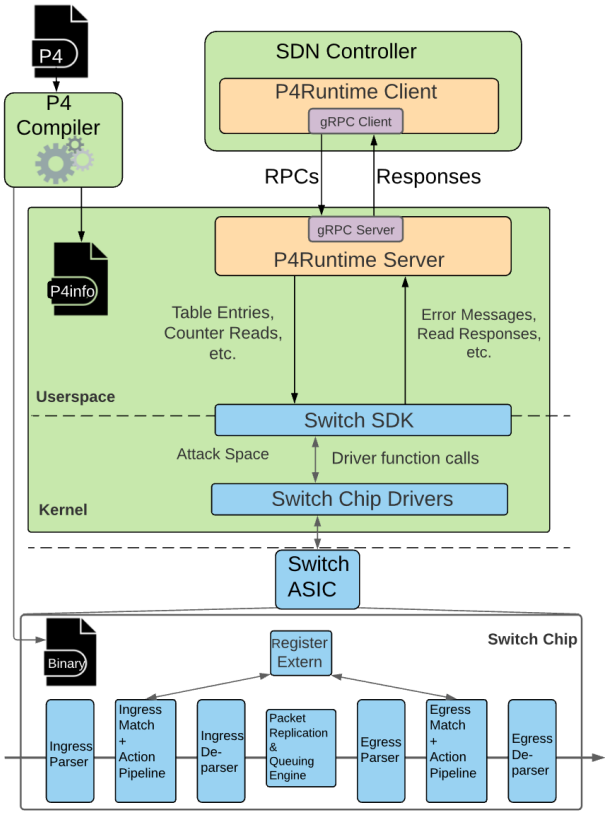

Fig. 1. P4-SDN Architecture

communication using Transport Layer Security (TLS). The idealised workflow of P4RT can be summarised as follows:

1) An architecture-compliant P4 program is compiled to produce a target-specific binary program to be run on the switch, and a P4Info file. P4Info is a contract between the DP and CP specifying information about the P4 program entities (e.g. table entries) present in the code that the controller can read or edit.
2) The P4RT client uses a *SetForwardingPipelineConfig* remote procedure call (RPC) to push both the P4Info file and binary program to the P4RT Server on the switch.
3) The target device installs the binary on the switch and stores the P4Info file within the switch operating system (OS) as a reference to the program so that the P4RT Server can service the requests of the Client.
4) The P4RT client monitors P4 entities by issuing *Read* RPCs and edits them using *Write* RPCs. The Client can also read or change the forwarding configuration at any time using the *GetForwardingPipelineConfig* and *SetForwardingPipelineConfig* RPCs, respectively.

C. gRPC Server - Switch Chip Communication

While the P4RT protocol serves an important purpose in that it defines a common interface by which any conforming P4 DP device can be controlled by a CP, it does not define the mechanisms by which the behaviours mandated by the protocol are implemented on any given switch. The exact

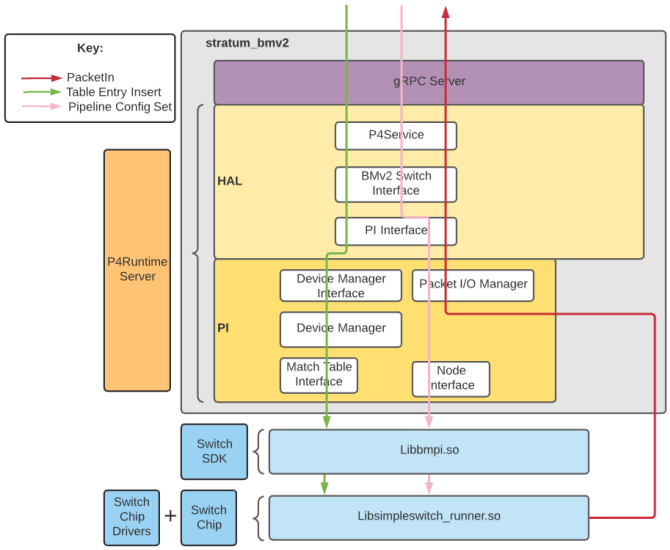

Fig. 2. Stratum-BMv2 Architecture

nature of this behaviour is device-specific (due to variances in OSs and hardware), but must always translate Protobuf messages received at the gRPC server to function calls to the switch software development kit (SDK). This SDK is specific to the individual switch chip and provides a set of API functions that allow other programs running on the switch OS to communicate with the switch chip's drivers.

In P4RT terms, the code that performs this translation to SDK calls is known as the P4RT Server. The PI (Program Independent) framework is currently the major attempt to provide a common P4RT Server implementation for programmable switches that support the PSA switch architecture.

D. Stratum/BMv2

BMv2 is a software switch designed by the maintainers of the P4 language as a reference P4 switch implementation. Despite BMv2 not being performant enough to be used in production, it is used extensively as a testbed for P4 programs.

Stratum is an open-source switch OS designed for use on whitebox P4 switches in SDNs. Stratum currently supports 8 different forwarding devices (including ASICs and software switches) [2] and incorporates management of all aspects of SDN switch behaviour, exposing three APIs to the CP for this purpose: P4RT for the management of a P4-programmable switch chip, gNMI to manage device configuration and gNOI for operations management. In this work, we are concerned only with P4RT. The key abstraction used in Stratum to implement this behaviour is the Hardware Abstraction Layer (HAL), which serves as an interface that converts P4RT RPCs to function calls to a device-specific implementation of a P4RT Server. For Stratum and BMv2, the PI framework is used as the P4RT Server implementation.

The BMv2-specific implementation of Stratum follows the structure shown in Fig. 2. The PI function calls to the forwarding device are implemented by the *Libbmpi.so* shared library and the functionality of the BMv2 switch is packaged into the *Libsimpleswitch_runner.so* shared library. The rest of the Stratum code runs as a Linux executable named *stratum_bmv2*, which calls the relevant shared library functions when required. This structure is reflected in Fig. 2, which also demonstrates the function call chain for common P4RT RPCs.

III. Attack Methodology

A. Threat Model

As previously noted, the objective of this work is to determine how an attacker with the ability to run their code on a P4 switch within an SDN may materially affect the behaviour of the network without being detected. We assume that the attacker is able to intercept and edit data to/from the controller. In particular, we assume that the attacker can inject their code before calls to the switch SDK or drivers (highlighted blue in Fig. 1), allowing them to edit the arguments passed to the SDK or driver functions. This threat model is more specific than many ADPV solutions, such as [30] which assumes that an attacker can drop, inject or delay packets without specifying how this may be achieved. It is also less strong than other threat models, such as that of [33], which additionally assumes that an attacker may exploit a side channel to exfiltrate sensitive information from packets.

The method of achieving compromise is out of scope as it is likely to depend on the vulnerabilities of an individual switch or network, but previous work has shown how SDN switches may be compromised by a malicious insider [28] or by exploiting switch-specific vulnerabilities [34]. In addition, it has been shown that supply chain attacks [21] or watering hole attacks on device drivers [35] may be used to achieve a compromise that fits with our threat model.

B. Attack Setup

To recreate the attack scenario, we set up a BMv2 (Behavioural Model version 2) software switch running the Stratum switch OS, controlled by the ONOS (Open Network Operating System) SDN controller. For each attack, we used the LD_PRELOAD trick [12] to preload a malicious shared library before the *Libbmpi.so* and *Libsimpleswitch_runner.so* shared libaries. This method (used by rootkits such as Jynx2 [8]), allows us to edit the arguments passed to the simulated ASIC's SDK and driver functions to suit the goals of our attack, as detailed in Section IV. Note that the use of BMv2 here relates only to the ability to intercept calls to its open-source SDK and driver shared library functions and does not depend on its inability to forward packets at line rate.

To test the ability of our attack code to alter network operations, we run an implementation of the P4Knocking application [37] on the BMv2 switch. P4Knocking is a P4 implementation of the port knocking stateful firewall method, where access is allowed through a given switch, only if each host requesting access firstly sends a series of packets with the correct sequence of TCP destination port numbers. The application is implemented using a P4 register, which stores an integer from 0 to 3, indicating the number of consecutive packets that have been received from a given IP address that

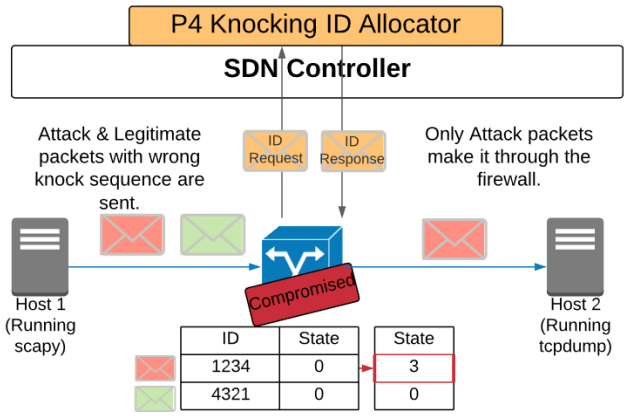

Fig. 3. Attack Flow Diagram

satisfy the secret knock sequence of TCP port numbers. Once the register value for an IP address is equal to 3, then traffic from that IP address is allowed to pass through the firewall. In addition to this, in order to reduce the memory occupied by the register, the program uses an optimisation where a 16-bit ID is used to index the register array rather than 32-bit IP addresses. These IDs are assigned by the CP (via the insertion of a table entry) upon receipt of a *PacketIn* from the DP, as depicted in Fig. 3.

The success of each attack was evaluated by two metrics: (1) the ability of the attacker to bypass the P4Knocking firewall and (2) the inability of the network operator to determine that an attack is taking place. Changes in forwarding behaviour were verified using two Mininet hosts attached to the P4 switch: one sending packets through the switch using scapy, and the other running tcpdump to detect packets that had bypassed the firewall (as shown in Fig. 3).

For each experiment, after verification of correct P4Knocking behaviour for legitimate packets, a series of TCP SYN packets were sent with the IP address set to that of the attacker, and with random TCP destination port values that did not follow the knock sequence. If the attacker successfully changed the program behaviour, these packets would be visible in the tcpdump output. (metric 1).

To assess the network operator's ability to detect the attacks, the logs of the ONOS controller were examined as a proxy for the CP's view of network operations. Any errors present in these logs would act as an indicator that an attack is taking place, thus constituting a failure of our attack. In addition, for the experiments involving changes to the switch program, the P4RT shell program was used to send *GetPipelineConfigRequests* (the baseline ONOS implementation did not do this) to query both the P4 binary (a JSON config file in the case of BMv2) and P4Info text file that constitute the P4 program. If either of these showed the presence of our attack code, the attack was considered a failure (metric 2).

IV. Attack Experiments

As identified in Section III, the goal of the attacker is to gain unauthorised access to the network by bypassing the P4Knocking firewall. This can be achieved by manipulating the P4 table entries or changing the P4 program[3].

The experiment details and results are presented in the following subsections with an overview of the tests and results presented in Table I. To enable reproducibility, we release our code at https://github.com/conorblack/AdvExpP4DP.

A. Attack 1: Manipulating P4 Table Entries

1) Attack Explanation: The goal of this attack is to indirectly edit the value stored in the register array for a target IP address by changing the associated ID value to one that has already been assigned to another IP address. In this way, the attacker's IP gets the same access rights as the legitimate IP, as it is the ID value that is used as the index to read the P4Knocking register. Unlike attacks that involve altering the P4 program itself, this attack can be performed on the legitimate running P4Knocking program, but it does have some other disadvantages. The major disadvantage is the fact that unless the attacker knows an IP address that has already passed the knock sequence, there is a risk that the new ID assigned to the attacker will not grant additional access.

2) Functions Involved: To perform this attack, an attacker must intercept one of the functions implementing a TableEntry *Write* message within the switch and change the ID value being added to the table. The function chosen was *MatchTable::add_entry* from *Libsimpleswitch_runner.so*. As this function is called when any table entry is inserted, it was necessary to ensure that we would only overwrite entries applicable to our attack. This was achieved by filtering by IP address and action name before editing the data.

In addition, given that the CP can use TableEntry *Read* messages to query the keys and action data associated with a table, the attack must also intercept a function involved in reporting the edited table entries back to the P4RT Server and replace the attacker-inserted value with the value originally assigned by the controller. For this purpose, the *get_entries* function from *Libsimpleswitch_runner.so* was intercepted. This function returns all table entries associated with a table and so our attack code only replaced the entry data if the match key and function matched those of the attack.

3) Results: According to the criteria set out in Section III, this attack was successful, as the firewall bypass was achieved with no error messages. However, if the function intercepting *get_entries* is removed, the ONOS logs do report inconsistent table entries, which the controller immediately deletes.

B. Attack 2: Changing the P4 Program

1) Attack Explanation: The goal of this attack is to change the P4 program running on the switch to one that includes an exception to the P4Knocking firewall rules. There are several ways that the P4 program can be altered. However, we consider only an attack that changes both the P4Info file and switch program. The attack involves adding an additional table with

[3]The manipulation of P4 register values is another possible attack vector, but was omitted in our experiments as there was no implementation of reading and writing of register values in the PI framework at the time of writing.

TABLE I
COMPARISON OF ATTACK TECHNIQUES BY FUNCTIONS INVOLVED AND RESULTS

Attacks	Variants	Functions Involved				Results	
		_pi_table_entry_add	*MatchTable::add_entry*	*Context::add_entry*	*get_entries*	Attack Packets Forwarded?	ONOS Logs
Manipulating Table Entries	w/ Read intercept	×	Intercepted & Called	×	Intercepted & Called	✓	No Errors
	w/o Read Intercept	×	Intercepted & Called	×	×	✓	Inconsistent Table Entries
Changing P4 Program	Controller-initiated	Intercepted & Called	×	Intercepted & Called	×	✓	No Errors
	Attacker-initiated	Called	×	Intercepted & Called	Intercepted	×	RPC Timeouts

a single action that automatically sets the knock state of any IP addresses added as match keys to 3, thus allowing any packets from these IP addresses to to pass through the firewall. The advantage of this approach from an attacker's perspective is the additional configurability, given that it allows an attacker to change the target IP address.

As an additional variant to this attack, we attempt to change the switch program both when the CP initiates a change and at a time when the CP is not expecting it. A successful pipeline change in the latter case would give an attacker total flexibility over when the attack took place. Otherwise, they would have to wait for the controller to change the switch program to launch their attack, something which may never happen for some switches whose operation remains fixed over time.

2) Functions Involved: The function that needs to be called to install a new switch program is *_pi_update_device_start* from *Libbmpi.so*, which takes both the P4 program JSON and the P4Info file (as a C++ struct) as arguments. For our attack, the arguments are changed to our attacker's version of these. In addition, a function needs to be intercepted that allows us to write table entries to our extra switch table. As our table does not exist in the legitimate P4Info file, we cannot use any functions from *Libbmpi.so*, as these use the legitimate P4Info file as a frame of reference and so do not recognise our additional table ID. As all of the arguments need to be forged, we intercept the *add_entry* function from the Context class, as this takes string arguments, unlike the *MatchTable::add_entry* function intercepted in Attack 1.

To carry out the attack variant where the P4 program is changed without CP initiation, the *get_entries* function was again intercepted, and the *_pi_update_device_start* function called from within it. This function was chosen as it is called frequently, ensuring our program will be loaded.

3) Results: According to the criteria set out in Section III, the main attack variant was successful, as the firewall bypass was achieved with no error messages. However, once the P4 program is changed from the *get_entries* function, the ONOS logs show that the switch has stopped responding to *Read* and *Write* RPCs, indicating that the switch has gone offline. The switch never recovers to forward any packets, thus this attack variant fails against both success criteria.

V. DISCUSSION

A. Findings

Our first key finding from the experiments is the ability to modify table entries to alter stateful behaviour without detection. This capability must be considered in future ADPV solutions, as current solutions do not address it. However, one of the interesting findings when implementing this attack was the aggressiveness of the ONOS controller in deleting the attacker-inserted table entries when they were not spoofed by intercepting calls to the *get_entries* function. While spoofing these table entry reads in the software switch was easily achieved, the extra burden of this on a hardware switch could be significant. Given that Barefoot's Tofino switch boasts of being able to add 100,000 new flow rules per second [4], the need to intercept each of these writes and any associated reads to find and edit relevant table entries may add such significant processing overhead to cause delays or even crashes.

A second key finding is the ability to intercept *SetPipelineConfigRequests* such that a different program can run on the switch than was intended by the network operator. Furthermore, the only restriction on the structure of this program is that it includes at least all of the P4 entities present in the legitimate program, allowing an attacker to use 'ghost' tables in the DP in their attack. Despite the fact that this attack only succeeds if the CP triggers a program change at the switch, once the attacker's program has been installed, it requires no additional maintenance to keep it hidden from the controller. This is because the P4RT Server that communicates with the controller stores a copy of both the switch program and the associated P4Info file locally and uses both to respond to *GetPipelineConfigRequests* from the controller and as the point of reference when querying the values of P4 entities from the switch chip. From a defence perspective, this greatly diminishes the usefulness of using *GetPipelineConfigRequests* and Table *Reads* in detecting attacks, as there is no requirement for an attacker to spoof the values returned by these requests.

The generalisability of the program change attack to hardware targets will vary by device as with the other attacks. However, the responsibility for the implementation of the P4RT Requests lies with the PI library, which can not only be used directly by switch vendors, but also serves as a reference for proprietary P4RT Server implementations.

B. Potential new approaches to ADPV

The implications of these findings on the development of future ADPV solutions are twofold. Firstly, the success of the attacks in manipulating stateful data structures in P4 programs confirms that this must be taken into account in the development of future ADPV mechanisms applicable to P4 DPs. Secondly, the lack of natural defence mechanisms in

P4RT sets a high bar for future ADPV solutions, as the full range of programmability may be leveraged by an attacker to create more subtle attacks than changing OF table entries. A logical solution to these new attacks is to extend current ADPV mechanisms to take account of DP state. However, the explosion in state space that would have to be covered by stateful extensions to probe-based defences greatly increases their complexity, while potentially reducing their benefit. For redundancy-based defences, the logical extension would be to replicate the calculation of state variables across many switches. The limitation in this case is that the replication of state requires additional operations in the switch, such as writing to registers, which may place additional processing time onto each packet. Furthermore, the potential use of additional memory to store state variables may be limited by the amount of memory available, which is often scarce on high-performance ASICs. The concept of replicating state is also in direct contrast to other state-of-the-art work, such as SNAP [3] and Poseidon [39], where stateful applications have been distributed across several switches to ensure memory and latency limits aren't exceeded.

As an alternative ADPV approach, in our further work, we propose to exploit the programmability of the switches to implement a moving target defence that frequently changes the expected behaviour of DP switches. For instance, a CP could dynamically change the switch from which counter statistics are requested, making any attack code obsolete as its statistics are no longer used by the CP application. The major advantage of this approach is the ability to place the burden on the attacker to constantly innovate their attack, which may not always be feasible (particularly if the attacker does not have continuous access to the switch) as it requires the attacker to learn the nature of new switch programs and redesign their attack accordingly. A key consideration in developing this solution will be to address the burden of regularly changing the switch program on every DP switch for the method to be effective. This is crucial to enable the CP to maintain an up-to-date view of expected network behaviour and to minimize switch downtime while switching between programs[4].

Finally, it remains an open question as to whether a hardware root of trust (e.g. a Trusted Platform Module) could be used on switches to verify the integrity of P4 programs. Such a solution would likely be advantageous due to its limited burden on the network operator and universal coverage of P4 programs, regardless of statefulness. However, to our knowledge, the feasibility of this has not been investigated and other hardware-based switch integrity checking mechanisms have nevertheless been shown to be exploitable [18].

C. Limitations

We acknowledge that our experiments have been performed on a software switch not intended for production and that we have assumed that an attacker is able to gain access to a switch. We have provided justification for these conditions. Nevertheless, the only fundamental obstacle in achieving the aims of the presented attacks was the inability to change the P4 program running on the switch without it having to come offline (Attack 2, version 2). While this is an expected switch behaviour during configuration update, in this particular attack version, the switch never recovered from the errors to forward packets again. If this were to be replicated in production-grade switches, it would make this route of attack infeasible. However, it is unclear to what extent a production switch would replicate this behaviour.

[4]A switch downtime of up to 50 ms is required for the Barefoot Tofino, even when their Fast Refresh technology is used to minimise disruption [4].

VI. Related Work

As alluded to in Section I, current ADPV solutions fall into three major categories. REV [40] is a cryptographic solution that involves embedding Message Authentication Codes (MACs) into data plane packets to prove that they have traversed the switch. These cryptographic operations, however, limit throughput, and only verify packet traversal, not editing. Chiu et al. [11] propose sending crafted probe packets to an OF switch to trigger multiple match rules at once, verifying that they are applied correctly. The major limitation of this and other probing solutions is that they cover only pre-installed, legitimate flow rules and overlook additional attacker-installed rules. Finally, Preacher [33] is an ADPV mechanism that leverages redundancy, collecting the hashes of packets that traverse each switch at the control plane and verifying that the paths match predetermined expected traversals. This method does not account for stateful packet processing.

Aside from verification, other work has looked at the vulnerability of P4 and other SDN-ready DP switches. Dumitru et al. [14] investigate how bugs in P4 programs on different switches may be exploited by an adjacent attacker to crash the switch or exfiltrate data. Thimmaraju et al. [34] demonstrate how vulnerabilities in OvS were able to lead to malicious remote code execution on the switch. Finally, Pickett [28] shows how an attacker may use vulnerabilities in popular whitebox switch OSs to compromise and stay persistent in these switches.

VII. Conclusion

The fundamental motivation behind the development of ADPV solutions is the idea that the control plane may not always have the true picture of the state of the DP if malicious actors have control of DP switches. While this threat model has been assumed by current ADPV solutions for stateless Openflow-based DPs, in this work, we have demonstrated through several attack examples that this threat model can be extended to include P4 DPs. Furthermore, we identify that the required interventions of the attacker are fewer than what may have been expected. These findings motivate a new approach to ADPV that can incorporate P4 programmability and statefulness. We have highlighted the inadequacy of current solutions to address this expanded threat model and propose a potential new approach to secure P4 DPs against attacks. We will explore this approach in our future work.

References

[1] Barefoot Tofino Product Brief. https://barefootnetworks.com/products/brief-tofino. Accessed: October 2020.

[2] Stratum 20.09 Release Notes. https://github.com/stratum/stratum/releases/tag/2020-09-30. Accessed: October 2020.

[3] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 29–43, 2016.

[4] A. Bas. Leveraging Stratum to Achieve Fast P4 Program Swap on Tofino. *Presented at ONF Connect 2018*.

[5] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[8] R. Carbone. Malware Memory Analysis of the Jynx2 Linux Rootkit (Part 1): Investigating a Publicly Available Linux Rootkit Using the Volatility Memory Analysis Framework. Technical report, Defence Research and Development Canada Valcartier Quebec, Quebec Canada, 2014.

[9] T.-W. Chao, Y.-M. Ke, B.-H. Chen, J.-L. Chen, C. J. Hsieh, S.-C. Lee, and H.-C. Hsiao. Securing data planes in software-defined networks. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 465–470. IEEE, 2016.

[10] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei. How to detect a compromised SDN switch. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6. IEEE, 2015.

[11] Y.-C. Chiu and P.-C. Lin. Rapid detection of disobedient forwarding on compromised OpenFlow switches. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 672–677. IEEE, 2017.

[12] R. Cieslak. Dynamic linker tricks: Using ld_preload to cheat, inject features and investigate programs. *Retrieved March*, 12:2015, 2015.

[13] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Ndss*, volume 15, pages 8–11, 2015.

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

[15] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[16] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. FLOWGUARD: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 97–102, 2014.

[17] A. Kamisiński and C. Fung. Flowmon: Detecting malicious switches in software-defined networks. In *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, pages 39–45, 2015.

[18] J. Kataria, R. Housley, J. Pantoga, and A. Cui. Defeating cisco trust anchor: A case-study of recent advancements in direct {FPGA} bitstream manipulation. In *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 15–27, 2013.

[20] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford. Tracking p4 program execution in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 117–122, 2020.

[21] M. Lee and H. Moltke. Everybody Does It: The Messy Truth About Infiltrating Computer Supply Chains. *The Intercept*, 2019.

[22] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 490–503, 2018.

[23] N. P. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese. Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep.*, 2016.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[25] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini, and M. Tornatore. Machine-learning-assisted DDoS attack detection with P4 language. *IEEE International Conference on Communications*, 2020.

[26] 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*, pages 1–7, 2018.

[27] P4 Language Consortium. P4 runtime specification. *Website, https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html*, 2017.

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

[29] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig. SDNsec: Forwarding accountability for the SDN data plane. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–10. IEEE, 2016.

[30] T. Shimizu, N. Kitagawa, K. Ohshima, and N. Yamai. WhiteRabbit: Scalable software-defined network data-plane verification method through time scheduling. *IEEE Access*, 7:97296–97306, 2019.

[31] 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 & ML*, pages 1–7, 2019.

[32] 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*, pages 518–532, 2018.

[33] K. Thimmaraju, L. Schiff, and S. Schmid. Preacher: Network policy checker for adversarial environments. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 32–3209. IEEE, 2019.

[34] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid. Taking control of sdn-based cloud systems via the data plane. In *Proceedings of the Symposium on SDN Research*, pages 1–15, 2018.

[35] Vectra Networks. *Vectra Networks Discovers Critical Microsoft Windows Vulnerability that Allows Printer Watering Hole Attacks to Spread Malware*. 2016.

[36] R. Wang, D. Butnariu, J. Rexford, et al. Openflow-based server load balancing gone wild. *Hot-ICE*, 11:12–12, 2011.

[37] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger. P4Knocking: Offloading host-based firewall functionalities to the network. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 7–12. IEEE, 2020.

[38] C. Zhang, J. Bi, Y. Zhou, J. Wu, B. Liu, Z. Li, A. B. Dogar, and Y. Wang. P4DB: On-the-fly debugging of the programmable data plane. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[39] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches. In *Proceedings of NDSS*, 2020.

[40] P. Zhang, H. Wu, D. Zhang, and Q. Li. Verifying Rule Enforcement in Software Defined Networks With REV. *IEEE/ACM Transactions on Networking*, 28(2):917–929, 2020.

[41] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu. Foces: Detecting forwarding anomalies in software defined networks. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 830–840. IEEE, 2018.

[42] Y. Zhou, J. Bi, T. Yang, K. Gao, C. Zhang, J. Cao, and Y. Wang. KeySight: Troubleshooting Programmable Switches via Scalable High-Coverage Behavior Tracking. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 291–301, 2018.