# State4: State-preserving Reconfiguration of P4-programmable Switches

Ji, C.; Kuipers, F.A.

**DOI**
10.1109/NetSoft57336.2023.10175468

**Publication date**
2023

**Document Version**
Accepted author manuscript

**Published in**
2023 IEEE 9th International Conference on Network Softwarization

# State4: State-preserving Reconfiguration of P4-programmable Switches

Chenxing Ji, Fernando Kuipers
*Delft University of Technology*
{c.ji, f.a.kuipers}@tudelft.nl

***Abstract*—To cater to constantly changing network needs, enabling stateful reconfiguration of Network Functions (NFs) is crucial. Recently, there has been growing interest in offloading NFs to programmable network devices. Unfortunately, it is currently not possible to maintain the full state of NFs during a switch reconfiguration without consuming network resources from and to neighboring switches. In this paper, we present State4, a framework that maintains the state of P4 programs during the reconfiguration of a P4-programmable network device, by only using a small amount of local resources on the switch undergoing reconfiguration. State4 acts on both the in-switch control-plane and the data-plane. By utilizing the in-switch local controller, State4 requires no external network resources to achieve reconfiguration while preserving states. As such, State4 enables on-the-fly reconfiguration of stateful NFs, at minimal traffic disruption, where previously traffic had to be re-routed.**

***Index Terms*—Stateful Data-plane, Programmable Data-plane Virtualization, Reconfiguration**

## I. Introduction

The emergence of the Programming Protocol-independent Packet Processor (P4), a domain-specific language used for programmable network devices, has accelerated research on programmable data-planes [1]. Compared to traditional network switches, programmable switches enable new protocols and functionalities to be deployed rapidly without the need to wait for vendor-specific implementations.

Together with the development of the programmable data-plane, there has also been significant interest in offloading Network Functions (NFs) traditionally run on servers to the programmable data-plane (e.g., load-balancers [2], [3], stateful firewalls [4]–[6], or network storage [7]). However, programmable network devices have been designed to support fast packet forwarding and struggle to provide multiple network functionalities over a single pipeline. Yet, since single NF applications often do not take up all the available resources of a switch, it is desirable to run multiple NFs on a single programmable network device for cost and efficiency reasons. In an attempt to accommodate multiple NFs on a single switch, researchers have been leveraging virtualization techniques.

In general, programmable data-plane virtualization frameworks take either a hypervisor approach [8]–[10] or a compiler-based approach [11], [12]. The hypervisor approach typically loads a generic hypervisor P4 program and manipulates table rules to allow NF loading and unloading at runtime. While the hypervisor approach can preserve runtime states, such as table rules, it does not support stateful NFs, because the hypervisor cannot resize and remap registers at runtime. On the other hand, compiler-based approaches use static analysis, which allows them to combine programs and reload the pipeline with a new configuration. The compiler-based approach allows for more efficient use of resources than the hypervisor approach, but the reconfiguration of the pipeline wipes all the state and hence calls for specific procedures to maintain that state. This means saving the runtime state and redeploying it after the reconfiguration. However, due to the high throughput of programmable network devices, the in-switch state can change rapidly, making it challenging to recover the previous state after a pipeline reconfiguration.

Our framework, State4, addresses this challenge. State4 scales well since its resource usage is independent of the number of allocated stateful resources. State4 utilizes a control-plane plus data-plane approach. First, it inserts P4 code to clone packets that alter state and send the cloned packets with state information to a local controller. Next, it uses the controller to query the state from the control-plane to reduce overhead. Then, it combines both the cloned state information and control-plane information to achieve state synchronization after reconfiguration. Our main contributions are as follows:

1) We propose State4, a framework[1] that adds support for compiler-based stateful NF reconfiguration of P4-programmable network devices.
2) Our framework includes a mechanism that minimizes traffic disruption during reconfiguration.
3) We implement and evaluate a prototype of our framework with `bmv2` in Mininet.

## II. Motivation

Currently, there are two main approaches for enabling virtualization on programmable data-plane devices: (1) the hypervisor approach and (2) the compiler-based approach. While the compiler-based approach allows for more efficient use of resources than the hypervisor approach, it is also presently unable to maintain the functionality of stateful NFs during a switch update or switch reconfiguration.

While previous research addressed offloading NFs to the programmable data-plane, applying multiple NFs within a network requires multi-tenancy and dynamic resource allocation, which the programmable data-plane currently cannot provide

[1]The source code of our State4 prototype is available at https://gitlab.tudelft.nl/lois/State4

at runtime. Therefore, considerable research has been devoted to applying virtualization techniques to programmable data-planes [8]–[13].

In the remainder of this section, we will explain why we focus on aiding the compiler-based approach over the hypervisor approach. Subsequently, we will discuss the inadequacy of the current approaches to achieve state synchronization after a reconfiguration.

### A. Hypervisor vs Compiler-based

The hypervisor and compiler-based approaches impact the statefulness and the runtime resource usage differently:

**Statefulness** Currently, one of the main differences between the hypervisor and the compiler-based approaches is their (in)ability to keep state. With the hypervisor, the hypervisor program handles adding functionalities to the data-plane by adding or removing match-action table entries. Through this, it is easy for this approach to maintain the state of P4 programs while modifying functionality at runtime. On the other hand, because the compiler-based approach reconfigures the pipeline to update NFs, it clears the runtime state.

The hypervisor approach, however, also has issues in supporting stateful P4 programs. Even though some states can be maintained while changing (at runtime) P4 functionalities, estimating the number of resources and (re)allocating registers for each NF is currently infeasible through the hypervisor programs [8], [9], [14]. This typically leads to over-provisioning and hence inefficient use of the precious resources on a switch.

**Resource Usage** Compared to commodity servers with gigabytes of memory that can easily host NFs, programmable data-plane devices typically only have SRAMs and TCAMs in the magnitude of megabytes [15]. This makes resource efficiency highly crucial on such devices.

The hypervisor approach typically allocates additional match-action tables for mapping NF programs during runtime. For instance, Hyper4 [8] pre-allocates match-action tables for emulating all possible actions, while HyperVDP [9] allocates dozens of match-action tables and utilizes condition-checking to apply the tables. Both approaches result in low utilization of allocated resources. For example, Hyper4 [8] allocates 13 tables at runtime for an L2-forwarding functionality, whereas a simple L2-forwarding program only requires 2 forwarding tables. Although follow-up works have reduced that inefficiency [10], [11], their resource utilization is still significantly worse than that of a native merge. Hence, hypervisor approaches are inefficient in terms of resource utilization.

For the compiler-based approach, the required amount of match-action tables is determined by the NFs needed. In addition, a compiler-based approach can also perform an analysis of the NFs to determine the best shareable resources, leading to reduced resource usage.

Unfortunately, updating functionalities for compiler-based approaches comes at the cost of traffic disruption during a pipeline reconfiguration. Although previous research has explored pipeline manipulation on program reload to mitigate this problem [13], this approach requires the help of an additional pipeline, which lowers the resource utilization of the network. In addition, state synchronization between the original and the target pipeline causes state inconsistency. Given the crucial importance of resource efficiency and the apparent benefits of the compiler-based approach, we have set out to overcome the existing statefulness hurdle in that approach, instead of working to improve a hypervisor.

### B. State Synchronization

Maintaining state correctness is crucial for stateful NFs, as they rely on those states to provide their functionality. For example, consider a network with a port-knocking firewall. The connection has been established between a client and a protected server. If the in-switch state is lost during a reconfiguration, the switch would drop the flow that was previously going through, because the firewall rule no longer recognizes the flow. Existing state-of-the-art solutions [13], [16] aim to tackle this problem through different approaches:

P4NFV [13] takes a controller approach to query runtime state. However, the querying time for stateful components increases up to seconds when there is a large number of states. Therefore, simply querying the state of the NFs of a switch using the control-plane would lead to state inconsistency. For example, consider a packet that modifies the current NF state after the reconfiguration query of the same stateful component.

Swing State [16] uses the data-plane for achieving state synchronization by cloning and sending state information from one pipeline to another. Hence, Swing State may suffer from packet loss and takes up additional network resources (on and toward the other pipeline). In addition, their work introduced hard and soft states to reduce the number of states needed for migration by only migrating hard states and by leaving it to the developer to write transfer functions for migration.

## III. Design Challenges

**Resource Constraints** The resources on a programmable network switch are usually quite constrained. For example, the Intel Tofino3 switch has 200 Megabits of SRAM and 10.3 Megabits of TCAM per pipeline [15]. Given the constrained resources of programmable switches, our framework should not demand significant additional resources (i.e., memory). The amount of NFs that can run on a programmable switch should ideally remain the same with or without the support for stateful NFs.

**State Consistency** Stateful network functions rely on the states stored to provide correct/accurate functionalities. If not, then for applications such as NAT [17] and port-knocking firewall [4], the established benign flows would be blocked/dropped. Moreover, during a switch reconfiguration, while maintaining state consistency, it is crucial to minimize any traffic disruption caused by the unavailability of NFs. Therefore, programmable switches need to forward packets in parallel with the state synchronization process, which – because of the fast-changing in-switch states – is challenging.

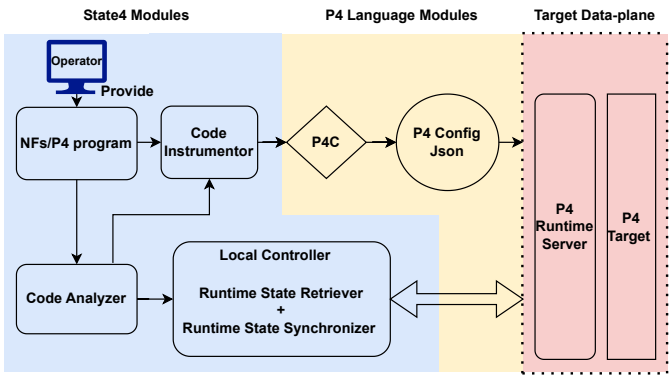



Fig. 1: Workflow of State4 for a switch reconfiguration.

## IV. Framework Design

We have designed a framework to overcome the design challenges mentioned in Section III. Our system consists of several modules (further explained below):

- Code Analysis Module (CAM): Scans the provided P4 program to retrieve stateful component information.
- Code Instrumentation Module (CIM): Modifies the provided new P4 program to perform packet cloning and send state change information to the local controller during reconfiguration.
- Runtime State Retrieving Module (RSRM): Reads the current switch state from the control-plane and receives state update information from the data-plane.
- Runtime State Synchronization Module (RSSM): Pushes the received state information and the new configuration to the switch.

Figure 1 depicts the workflow of a switch reconfiguration and how the individual modules use the information obtained by other modules. The operator provides a new P4 program that is scanned by CAM. CIM then uses the information gathered by CAM to modify the new P4 program. Hence, by using State4, any program installed on the switch will also have been instrumented by CIM and thus is able to clone packets and send state information to the controller. When a new program is to be loaded, it also triggers RSRM, which retrieves the runtime state from the "old" P4 program through both the control- and data-planes based on CIM's information about the P4 code. Last, RSSM analyzes the state retrieved by RSRM, pushes the state to the runtime of the new program, and performs the pipeline reconfiguration configuration swap. We utilize a local controller to reduce latency between the controller and the data-plane to minimize state inconsistency and avoid state inconsistency caused by potential packet loss. Our system defaults to preserving all states on a programmable switch. However, due to the cost of synchronization, State4 also allows the network operator to provide a list of stateful components such that only those components would be synchronized.

### A. Local Controller

State4, for the reconfiguration process, leverages a local in-switch controller instead of a remote controller elsewhere in the network. The local controller is directly connected to the CPU port of the switch, thus having a dedicated connection and low latency between the communication with the controller and the switch. For other control-plane functionalities, the local controller could either interface with a remote centralized controller or distributively synchronize with the other local controllers in the network.

### B. Code Analysis Module (CAM)

CAM handles retrieving stateful component usage information from the P4 source code provided by the network operator. CAM performs a one-time scan over the provided P4 code and identifies the name of the stateful components based on predefined P4 keywords for the provided architecture model, e.g., "registers." CAM creates a list of stateful components for every defined stateful keyword. The information is sent to CIM for inserting code with the associated components and to RSRM for reading the current runtime state.

CAM collects information about where the register write operations are executed and calculates the following: the total number of registers, the maximum index size of all the registers, and the maximum possible value any register index could represent. These values are used by CIM to compute the size of metadata needed for storing state information.

Since read access does not affect the current state of the P4 program, CAM does not need to record read access usage. Write access, on the other hand, does affect the current state and needs to be recorded by CAM. Therefore, for each write access to stateful components, CAM records the following three parts for the code instrumentation module: the name of the register components, the variable name of the register index to access, and the variable name of the value to write. The information is then used by CIM for storing corresponding values in the metadata for cloning and sending the updated states during the reconfiguration phase.

### C. Code Instrumentation Module (CIM)

CIM deals with the potential state inconsistency caused by reconfiguration (because states read by the controller may not be consistent with the state at the configuration swap point). CIM uses the information gathered by CAM to modify the code such that only the packets with state updates will be cloned to minimize the impact on the traffic. Listing 1 gives an example of the code inserted by CIM to a port-knocking firewall application. The details of each addition are explained in the remainder of this subsection. Note that all the code snippets inside Listing 1 are inserted into the original P4 program by CIM except for Lines 23-24, which is the write access to a stateful component in the original program.

We further describe CIM for each addition in the listing. Since packets are cloned inside the traffic manager, the cloned packets are only available during the egress processing. Therefore, state update information needs to be stored inside

```
1  /* Header code */
2  struct packetIn_meta {
3      bit<8> reg_num;
4      bit<24> index_num;
5      bit<80> value;
6  }
7  struct metadata {
8      @field_list(0)
9      packetIn_meta pkIn_meta;
10 }
11 @controller_header(''packet_in'')
12 header packet_in_header_t{
13     bit<8> reg_num;
14     bit<24> index_num;
15     bit<80> value;
16 }
17 /* Ingress code */
18 bit<1> clone;
19 register<bit<1>>((1)) clone_reg;
20 /* Ingress apply logic*/
21 clone_reg.read(clone,0);
22 /* Ingress state write to metadata */
23 pk_reg.write((bit<32>)meta.pk_meta.id,
24    meta.pk_meta.stage);
25 if (clone == 1) {
26    store_info_into_meta_2(8,
27       (bit<24>)meta.pk_meta.id,
28       (bit<80>)meta.pk_meta.stage);
29    clone_preserving_field_list(
30       CloneType.I2E,5,0);
31 }
32
33 /* Egress code */
34 action send_to_controller_with_data(
35    bit<8>reg_num,bit<24>index_num,
36    bit<80>value) {
37       hdr.packet_in.setValid();
38       hdr.packet_in.reg_num = reg_num;
39       hdr.packet_in.index_num = index_num;
40       hdr.packet_in.value = value;
41 }
42 /* Egress apply logic */
43 if (standard_metadata.instance_type == 1) {
44    send_to_controller_with_data(
45       meta.pkIn_meta.reg_num,
46       meta.pkIn_meta.index_num,
47       meta.pkIn_meta.value);
48    truncate(14);
49 } else {
50      hdr.packet_in.setInvalid();
51 }
52
53 /* Deparser code */
54 packet.emit(hdr.packet_in);
```

Listing 1: Example of inserted code in different P4 code blocks for a port-knocking firewall application. Comments depict the location of each line for a v1model program.

the metadata field to be preserved when reaching egress processing.

The CIM first inserts a struct inside the metadata field (Lines 2-10) and a custom header (Lines 11-16). The metadata field is used to preserve information across the ingress and the egress and the header is used as a `packet_in` for the controller. To represent all possible values for any given register with any index, we use the information obtained by CAM to calculate the number of bits needed for the maximum possible value. Both the custom header and the struct contain three variables to store information, the corresponding register number $n_{reg\_num} = \lceil \log_2 num\_registers \rceil$, the index of the register $n_{index\_num} = \lceil \log_2 max\_register\_size \rceil$, and the corresponding value $n_{value} = max\_index\_size$.

Then a 1-bit register is allocated (Line 19) at the beginning of the ingress processing to be the indicator of the stateful reconfiguration: `1=clone` and `0=idle`. The register value is controlled by RSRM to clone state updates of the NFs during the reconfiguration process. A temporary variable `clone` is also declared (Line 18) to store the value read from the register.

At the beginning of the ingress processing, CIM adds one line of code to read the allocated 1-bit register (Line 21) and stores the read value to an intermediate variable such that the entire ingress processing has access to read. This ensures a global state indicator for the P4 program to know whether the switch is undergoing a switch reconfiguration and whether information needs to be cloned when state-updating packets arrive. In addition, CIM also adds an action for storing state-write information into its corresponding field inside the created metadata structure.

CIM adds lines of code at each write access (Lines 23-31) to the state components for cloning purposes. The addition starts with checking the value read from the 1-bit register to determine the state. The variable name of the register component is used along with the state information to store in the declared metadata field. For the provided example, the number 8 in Line 26, indicates that this component is the 8th declared, which is the index value for the `pk_reg` component. Since we add the clone operation with a specific session identifier for forwarding the cloned packet to the CPU port, the information inside the metadata field is saved, and the program can access these values after the packet is cloned.

During egress processing, CIM first uses metadata to check whether the packet belongs to a cloned packet (Line 43), such that only the cloned packet needs to contain the information with write access, while the original packet gets forwarded without getting affected. Then, it adds an action (Lines 34-41) and applies the action (Lines 44-47) to copy information embedded inside the metadata field into the header structure for sending custom packets to the local controller. Finally, it uses the header structure's size to truncate the packet's size (Line 48) to minimize the amount of data between the data-plane and the controller. In addition, setting the added header valid allows the deparser to emit the header information correctly.

### D. Runtime State Retrieving Module (RSRM)

The communication flow among RSRM, RSSM, and the P4 switch is illustrated in Figure 2. RSRM handles receiving state information using both the control-plane and the data-

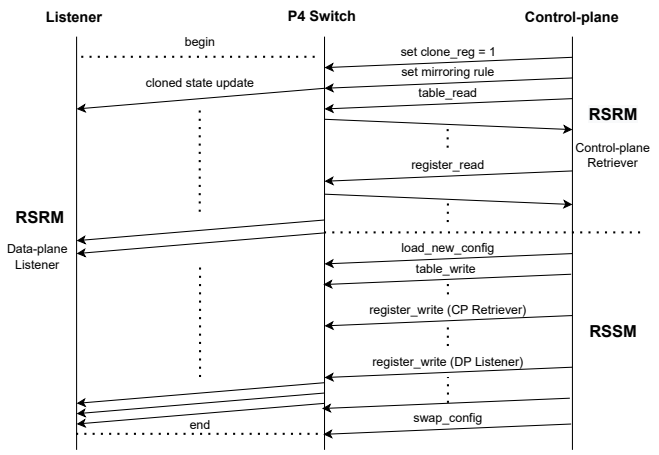



Fig. 2: Information flow among the programmable switch, the Runtime State Retrieving Module, and the Runtime State Synchronization Module.

plane. Thus, this module is split into two different controller modules: data-plane listener and control-plane retriever.

The data-plane listener waits for a `packet_in` event, an event defined by the P4Runtime specification [18], at the CPU port of the switch. In the meantime, the control-plane retriever utilizes the result of CAM and queries the control-plane to retrieve state information from the switch. The remaining part of this subsection describes the data-plane listener module and then explains the steps executed for the control-plane retriever. The data-plane listener starts before the control-plane retriever, ensuring no state information is lost. After receiving state information, the listener parses the received data, maps the state component number to the register name, and saves the state info for RSSM.

The control-plane retriever first sets the value of the 1-bit register allocated by CIM to one, indicating an `updating` state. Fine-grained instrumentation to each write access will enable sending the update state to the controller until the register value is reset to zero, i.e., when a pipeline is reconfigured. In addition, to enable the data-plane listener to retrieve cloned packets with state information, the control-plane retriever utilizes the `mirroring` operation specified by the Portable Switch Architecture (PSA) [19] to forward cloned packets to the CPU port of the switch.

The retriever then uses the collected names of the stateful components to query the current state, i.e., read all the indices of every register. Then, RSRM saves the received state information for RSSM to use.

### *E. Runtime State Synchronization Module (RSSM)*

RSSM handles three parts of the reconfiguration process. It (1) roughly synchronizes the state of the new program to match the state of the previous program using the RSRM received from control-plane querying, (2) applies a look-ahead algorithm to further synchronize the state information received from the data-plane cloned updates, and (3) reloads the programmable switch with the new P4 program.

*1) Synchronization of control-plane states:* First, the RSSM uses the states read by RSRM from the control-plane to achieve a rough synchronization between the new and old configuration. The overall state retrieved from the control-plane allows State4 to only clone the updates during the reconfiguration.

*2) Look-ahead synchronization:* For synchronizing data-plane cloned state updates, multiple messages could update the exact same index of the register. Consider a counter that counts the number of bytes over a single port, the counter value would be updated for every packet received. Thus, the RSSM would receive continuous updates with the same index that have incremental value. For such states, a new update would overwrite the previous outdated value, and only the newly updated value is needed for the data-plane. Thus, State4 adopts a look-ahead mechanism to reduce the number of states that need to be synchronized for the new program. The look-ahead mechanism checks if the next state to be processed operates on the same index of the same state component and, if so, skips synchronizing the current outdated state.

For reloading the programmable switch, we utilize the `SetForwardingPipelineConfig` Remote Procedure Call (RPC) from the P4Runtime specification [18] to configure the switch with a new P4 pipeline. RSSM utilizes two actions specified in this RPC: `VERIFY_AND_SAVE` and `COMMIT`.

The `VERIFY_AND_SAFE` action verifies whether the target can realize the provided configuration file. If the NFs require more resources than the target can provide, this action returns an error to inform the network operator. It also points any subsequent write requests to the new pipeline configuration, such that the NF state can be ready. This allows the NF to start processing right after the configuration swap happens. Another important factor about this action is that it does not modify the previous forwarding state. Thus, flows can still be processed up to this point, and traffic is not affected, and the state update happening within this process would be forwarded to RSRM to prevent state inconsistency.

After the `VERIFY_AND_SAFE` action is finished, RSSM reads the received data from RSRM to synchronize the states using the look-ahead mechanism through the control-plane. Since each state update during the reconfiguration is cloned and sent to the controller, compared to the state retrieved through the control-plane, the state information from the data-plane listener is guaranteed to be up to date. Therefore, these states from the data-plane listener are sent after all the data retrieved by the control-plane retriever. RSSM then reads the received state packets and writes each back to the switch. RSSM then uses the `COMMIT` action to realize the previously saved configuration file, and the pipeline is synchronized and configured to run the new NFs.

## V. Evaluation

In this section, we describe our prototype implementation and evaluate our proposed framework. Our experimental evaluation is conducted on a machine with an Intel® Core™ i7-9750H CPU at 2.60 GHz, 16 GB of memory, and Ubuntu 20.04 LTS. We used *CPUnetLOG* [20] for throughput measurements. We partially based our plot generation scripts on [21]. For the implementation of our proposed framework, we used Python3 in combination with the behavioral model version 2 (`bmv2`) [22], a software version of a P4 switch. To simulate the network, we use Mininet [23]. Our test topology consists of five hosts plus one server protected by a firewall. All of the hosts, including the server host, are directly connected to the bmv2 switch.

Our controller is implemented using a multiprocessing model such that the listener can perform simultaneously with the re-synchronization module. One thread acts as a listening controller and listens to the data-plane updates. Another thread retrieves the state from the control-plane and then parses the received information to re-synchronize the switch state. For the communication between the local controller and the control-plane, we leverage the `register_read` and `register_write` methods provided by `simple_switch_CLI` [22] to read and write to tables and registers. We used this implementation to evaluate State4 on the following aspects: the ability to preserve state, the state loss compared to the control-plane approach, and the cost of the mirroring operation.

### A. Evaluated NF

The choice of NFs for the evaluation is critical to reveal the impact and performance of State4. For the evaluation, the target bmv2 switch is configured for a realistic scenario with the following network functions: L2-forwarding, the HULA load-balancer [2], and a port-knocking firewall [4]. Since these programs are separate P4 programs and combining various programs is not the main challenge this paper is trying to solve, we manually combine these P4 programs into one program through chaining. The combined program is capable of performing the functionalities of all three programs. For the port-knocking firewall, state information is highly critical as each state change lost would lead to benign flows getting blocked by the application. Thus, we consider our choice of NFs to be a meaningful representation to evaluate State4 with.

### B. Stateful Reconfiguration Evaluation

This section shows that State4 supports full stateful reconfiguration and also evaluates how our framework impacts ongoing traffic. We conducted an experiment using the setup described above to show that State4 supports full stateful reconfiguration and that merely using a control-plane approach is insufficient for maintaining state. During the experiment, four flows that send one packet to the server every second were established. Among the four flows, one flow was directly forwarded by the firewall that started at 0s. The other three flows required port-knocking to establish the connections. Two

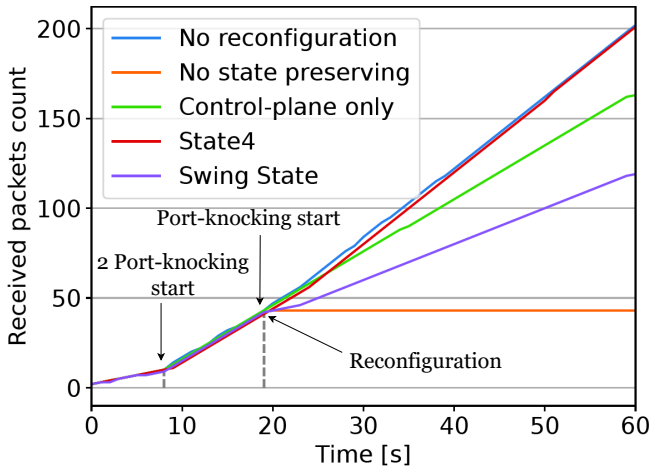



Fig. 3: Comparison of received packet count among State4, Swing State, a control-plane approach, and no state preservation.

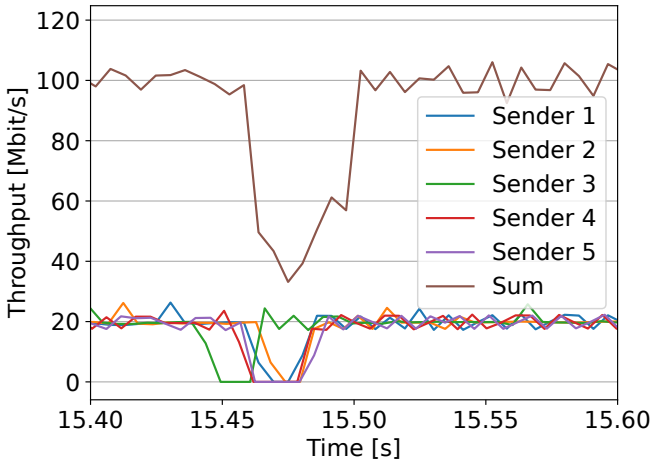



Fig. 4: 200 milliseconds interval zoom-in of the 30 seconds experiment.

of the three port-knocking flows started around 8s, and the third port-knocking flow started right before the switch configuration swap at around 19s. Our results for the experiment are shown in Figure 3.

When no state-preserving mechanism is applied, both table entries and register state are lost. As a result, all states are lost, and traffic cannot be received by the receiver. Swing State was only able to preserve and synchronize the state changes while in-network data-plane cloning was enabled. Therefore, Swing State requires constant cloning operations to be able to capture every state change. In Section V-D, we will explain why this form of data-plane cloning leads to significant overhead, and how State4 leads to a significantly reduced cloning overhead. On the other hand, pulling states from the control-plane also results in part of the register state information being lost.

Thus, one flow was dropped and the receiver was unable to receive all of the packets. Our results show that State4 outperforms both methods. The receiver obtained the exact packet count, indicating all state was successfully preserved after the reconfiguration. Figure 3 shows that State4 is able to preserve more states compared to the state-of-the-art Swing State [16] and the controller approach by the number of packets received from the port-knocking flows.

The performance impact evaluation pertains to 5 different hosts continuously sending packets through the P4 switch towards the receiving server. In addition, since our approach differentiates flows that only read the stateful components from flows that write to the in-switch states, we use three read-only flows and two flows that write state. Each of the flows is limited to 20 Mbps, such that they utilize a total of 100 Mbps. Our experiment lasts for 30 seconds and records the throughput every millisecond. The reconfiguration starts around 13.5s for around 2 seconds.

Our results show that for a P4 program with 236 KB of registers and seven tables, the total reconfiguration time taken is around 1.82 seconds. Figure 4 zooms in on the pipeline configuration swap phase during the experiment from 15.40 s to 15.50 s, from which it is clear that traffic disruption caused by the configuration swap of `bmv2` lasts for approximately 20ms. Flow states were also preserved after the disruption.

The traffic disruption shown inside Figure 4 is not introduced by our framework, but an inevitable disruption caused by the pipeline configuration swap on the `bmv2` switch. In these two experiments, State4 was able to preserve all the state.

### C. Delta-loss Evaluation

We measured and compared the delta-loss difference between our approach and the control-plane approach adopted by [13]. To find the exact state difference caused by the two systems, we collected the number of bytes transmitted over a single port and compared the number of transmitted bytes with the value stored inside the data-plane at the end of the transmission.

We used a register implemented as a counter to collect the number of bytes transmitted by the specific flow. We conducted the experiment with a flow at a transmitting rate varying from 10 Mbps to 120 Mbps that lasted for 30 seconds. Each experiment was conducted twice to compare the state divergence between State4 and the control-plane approach.

Figure 5 shows the difference in state loss percentages: State4 clearly outperforms the controller approach. Our results show that State4 led to less than 1% of state divergence, which we deem acceptable as applications utilizing such counter behavior typically apply sampling [24], [25] or approximation algorithms [26]–[28] that tolerate slight state divergence.

As shown in Figures 6a and 6b, State4 induces significantly less state loss than the control-plane approach. For the controller-based approach, shown in Figure 6a, the state inconsistency gap exists between the time that a particular state is read from the control-plane and the time that data-plane is

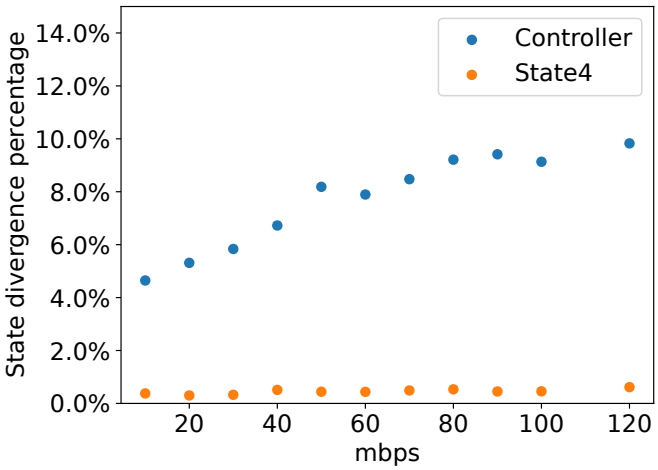



Fig. 5: Comparison using a packet counter of data-plane state divergence between the Controller approach and State4.

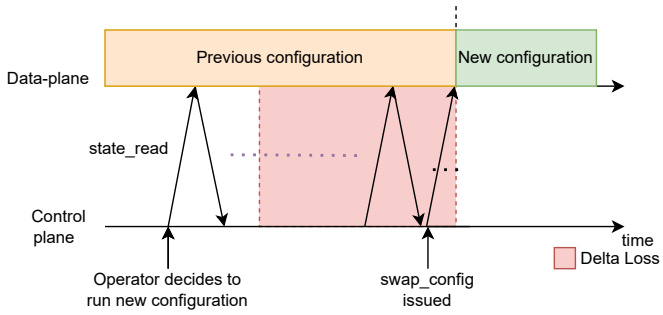
configured with the new configuration. As a result, all the state updates between component querying and new configuration are lost. Furthermore, since all components are queried from the control-plane, this approach does not scale well. As the number of state components increases, the increase in total pulling time yields a significant inconsistency in the data-plane state.

The state inconsistency introduced by State4 is minor. Since the control-plane listener and the data-plane processing operate simultaneously, the data-plane continuously clones and sends state updates to the listener while the RSSM processes all the received packets. Figure 6b illustrates the critical period for State4, during which the RSSM issues the configuration swap command before the configuration is swapped. State updates during this period would no longer be processed by the RSSM as the switch is already switched to the new configuration.

### D. Cloning Cost and Impact

Since data-plane cloning introduces overhead to data-plane processing, it is crucial that we examine the cost of the cloning operation and how this operation impacts the ongoing traffic. Furthermore, although Swing State can capture as many updates as State4 with data-plane cloning enabled, significant resource overhead and traffic impacts are introduced. Therefore, we also implemented the data-plane cloning operation of Swing State and compared our results with it.

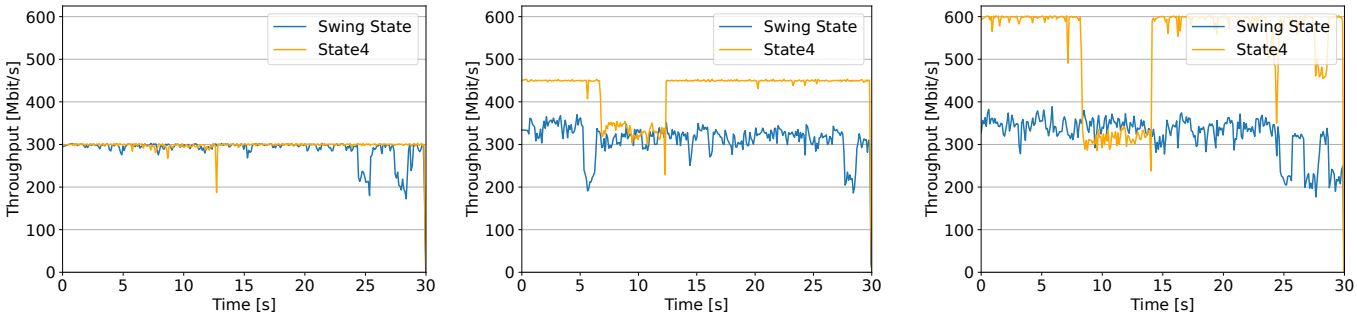
To test the cost of the cloning operation to the ongoing traffic on the `bmv2` implementation, we first find the maximum steady processing throughput of the P4 program. We used a single flow and found through experiments that our combined network function implementation is able to perform steadily at a maximum 600 Mbps. In addition, to identify the maximum cloning cost, we used a register implemented as a counter such that every packet processed is associated with a state write operation. Thus, every packet processed would be cloned and forwarded to the local controller during the reconfiguration




(a) State loss in the controller approach.

(b) State loss in the State4 approach.

Fig. 6: Illustration of state update loss for both controller approach and State4 approach.




(a) 300 Mbps throughput.

(b) 450 Mbps throughput.

(c) 600 Mbps throughput.

Fig. 7: Comparison of State4 and Swing State on the impact of the cloning operation, during the reconfiguration phase.

phase. We conducted the reconfiguration experiments with State4 in terms of flows going at 600 Mbps, 450 Mbps, and 300 Mbps, and compared the results of Swing State at the same throughput.

Figure 7 illustrates the impact of the cloning operation for both State4 and Swing State on the ongoing traffic. From Figure 7a, with traffic less than half of the maximum switch throughput, traffic was barely impacted by State4 during the entire phase except for the configuration swap at the switch. Figures 7b and 7c indicate that State4 outperforms Swing State when the throughput is over half of the maximum switch throughput. This is because of the cloning operation for every packet with state updates, which effectively halves the throughput at maximum utilization. Moreover, as State4 only clones the packets during the reconfiguration, State4 impacted traffic significantly less than Swing State [16], which requires constant data-plane cloning and forwarding such that maximum switch throughput is dropped to half.

In conclusion, we have evaluated the proposed State4 in terms of the ability to maintain state, the tolerable state divergence, and the introduced overhead. We argue that our combined control-data plane approach provides a better way than Swing State and the controller approach adopted by [13]. State4 minimizes state inconsistency caused by reconfiguration and causes a limited impact on existing traffic.

We do remark that the performance of our work in hardware switches has not (yet) been evaluated and is related to the processing capabilities of the local controller. The local controller should be able to keep up in terms of processing cloned packets given the relatively high forwarding speeds of the switch. In case the controller is not able to keep up, the resulting state divergence of our system would increase. To further improve the performance in such cases, approximation mechanisms can be applied (as suggested in [29]) such that the number of states generated from the data-plane cloning operation would be significantly reduced.

## VI. Related Work

In this section, we describe existing approaches for stateful reconfiguration and unveil their problems. Previous works [13], [16] have mainly focused on two approaches for state-preserving reconfiguration: pulling the state from the control-plane and sending state update information directly from the data-plane.

He et al. [30] tried to tackle the inconsistency problem by applying Control Flow Graph (CFG) pruning to reduce the number of state components to maintain for a given P4 program. However, the scalability problem persists for the control-plane approach when NFs request significant resources for one stateful component.

P4Consist [31] uses both the data-plane and the control-plane. However, their work targets the detection of inconsis-

tencies between the control-plane and the data-plane, which is orthogonal to our work.

FlexCore [32] was proposed to enable partial reconfiguration of the programmable device without disruption to the network. However, their work falls short in terms of reconfiguring stateful network functions due to the difficulties of porting the current state to the new program.

Wang et al. [33] proposed a hybrid approach for supporting multi-tenancy with programmable network devices in the cloud: they combine the hypervisor approach and the compiler-based approach. To enable multi-tenancy, they proposed to dynamically allocate a big register array to each tenant program. Since current hardware does not support partial reconfiguration, their work requires switches to reload, and it thus also suffers from state loss during a reconfiguration.

## VII. Conclusion

This paper has presented a framework for supporting stateful programmable switch reconfiguration that combines two methods: data-plane forwarding and control-plane querying. We have implemented a prototype and evaluated our proposed framework, called State4, under a realistic scenario and demonstrated that it only incurs negligible overhead to memory usage and leads to zero packet loss. Hence, by incorporating State4 into programmable data-plane virtualization systems, the systems can enable NFs to be updated on-the-fly on the programmable switch.

## VIII. Acknowledgement

This work is part of NExTWORKx, a collaboration between TU Delft and KPN on future telecommunication networks. We further thank Adrian Zapletal, Belma Turkovic, and Anup Bhattacharjee for their valuable feedback.

## References

[1] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, 2021.

[2] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable Load Balancing Using Programmable Data Planes," in *Proceedings of SOSR*, 2016.

[3] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz, "MP-HULA: Multipath Transport Aware Load Balancing Using Programmable Data Planes," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018.

[4] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger, "P4Knocking: Offloading host-based firewall functionalities to the network," in *Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020.

[5] M. Caprolu, S. Raponi, R. Di Pietro, and A. Antonopoulos, "FORTRESS: An Efficient and Distributed Firewall for Stateful Data Plane SDN," *Sec. and Commun. Netw.*, 2019.

[6] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful Network-Wide Abstractions for Packet Processing," in *Proceedings of SIGCOMM*, 2016.

[7] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of SOSP*, 2017.

[8] D. Hancock and J. van der Merwe, "HyPer4: Using P4 to Virtualize the Programmable Data Plane," in *Proceedings of CoNEXT*, 2016.

[9] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-Performance Virtualization of the Programmable Data Plane," *IEEE Journal on Selected Areas in Communications*, 2019.

[10] P. Zheng, T. Benson, and C. Hu, "P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs," in *Proceedings of CoNEXT*, 2018.

[11] R. Parizotto, L. Castanheira, F. Bonetti, A. Santos, and A. Schaeffer-Filho, "PRIME: Programming In-Network Modular Extensions," in *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[12] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4VBox: Enabling P4-Based Switch Virtualization," *IEEE Communications Letters*, 2020.

[13] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: An NFV architecture with flexible data plane reconfiguration," in *14th International Conference on Network and Service Management (CNSM)*, 2018.

[14] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing Dataplane Programs with P4," in *Proceedings of SIGCOMM*, 2020.

[15] Intel, "Intel® Tofino™ Intelligent Fabric Processors," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-product-brochure.html

[16] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes," in *Proceedings of SOSR*, 2017.

[17] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, and G. Bianchi, "Implementing advanced network functions for datacenters with stateful programmable data planes," in *LANMAN*, 2017.

[18] The P4.org API Working Group. (2020) P4Runtime specification, version 1.3.0. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[19] The P4.org Architecture Working Group. (2018) P4 16 portable switch architecture (PSA), version 1.0. [Online]. Available: https://p4.org/p4-spec/docs/PSA-v1.0.0.html

[20] M. Hock. (2018) CPUnetLOG. [Online]. Available: https://git.scc.kit.edu/CPUnetLOG/CPUnetLOG

[21] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, "Towards a deeper understanding of tcp bbr congestion control," in *IFIP Networking*, 2018.

[22] The P4 Language Consortium. (2016) BEHAVIORAL MODEL (bmv2). [Online]. Available: https://github.com/p4lang/behavioral-model

[23] "Mininet: Rapid prototyping for software defined networks," 2021. [Online]. Available: https://github.com/mininet/mininet

[24] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks," *SIGCOMM CCR.*, 2011.

[25] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI*, 2014.

[26] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *Proceedings of SIGCOMM*, 2016.

[27] S. L. Feibish, Z. Liu, N. Ivkin, X. Chen, V. Braverman, and J. Rexford, "Flow-level loss detection with $\Delta$-sketches," in *Proceedings of the Symposium on SDN Research*, 2022.

[28] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, "Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware," in *IFIP Networking Conference (Networking)*, 2020.

[29] X. Chen, H. Liu, Q. Huang, D. Zhang, H. Zhou, C. Wu, X. Liu, and Q. Yang, "Toward Low-Latency and Accurate State Synchronization for Programmable Networks," *IEEE/ACM Transactions on Networking*, 2022.

[30] M. He, A. Blenk, W. Kellerer, and S. Schmid, "Toward consistent state management of adaptive programmable networks based on p4," in *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies*, 2019, pp. 29–35.

[31] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid, "P4Consist: Toward Consistent P4 SDNs," *IEEE Journal on Selected Areas in Communications*, 2020.

[32] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime Programmable Switches," in *NSDI*. USENIX Association, 2022.

[33] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda, "Multitenancy for Fast and Programmable Networks in the Cloud," in *12th USENIX Workshop on HotCloud*, 2020.