Secure State Migration in the Data Plane

Jiarong Xing
Rice University

Ang Chen
Rice University

T.S. Eugene Ng
Rice University

ABSTRACT

Programmable data planes enable *stateful* packet processing at hardware speeds—a new capability central to many recent systems. However, protocols and systems that effectively manage data plane state remain underexplored. This paper considers the problem of *secure state migration*, which can serve as an important building block for state management tasks. It delivers data plane state from a source switch to a destination effectively without a software controller, while providing strong cryptographic guarantees on authenticity. Our protocol, P4Sync, tackles several technical challenges, such as adapting memory copy techniques in VM migration, offloading per-packet security operations to the data plane, and amortizing heavyweight cryptographic overheads over a batch of packets. Our initial validation shows that P4Sync has low traffic and memory overheads.

CCS CONCEPTS

• **Networks → Programmable networks**; • **Security and privacy → Security protocols**.

KEYWORDS

Programmable data planes, P4, State migration, Stream authentication, Network security

ACM Reference Format:
Jiarong Xing, Ang Chen, and T.S. Eugene Ng. 2020. Secure State Migration in the Data Plane. In *ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure (SPIN 2020) (SPIN '20), August 14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3405669.3405822

1 INTRODUCTION

A recent advance in networking technology is the development of *programmable data planes* [11]. Unlike traditional fixed-function switches, recent switch hardware can be reconfigured using high-level languages, such as P4 [5], PoF [49], and NPL [4]. Users can realize a wide range of packet processing behaviors that were previously only implementable in controller software. In addition to supporting customized headers and protocols, a prominent feature of programmable data planes is *stateful* packet processing. A typical programmable switch has O(10MB) stateful memory. A switch program can allocate register arrays from memory, and these registers can be updated on a per-packet basis. Moreover, as long as a switch program compiles successfully to a target, the program is guaranteed to run at Tbps linespeed.

Data plane programmability has enabled an array of tasks to run directly inside the network. Recent use cases include network measurement and monitoring [58], network security [28, 38, 56, 57], load balancing [25, 29, 39], and failure detection and diagnosis [19, 24]. Stateful packet processing is central to many of these in-network tasks. Depending on the specific application, the particular form of state may vary—e.g., some tasks rely on per-flow state for access control [28] or covert channel defense [56], other tasks keep per-destination state for load balancing [25, 29], and yet others may accumulate state per-packet for fine-grained monitoring [48]. In each case, the network data plane can change the packet processing behavior dynamically without having to consult a central controller. This stands in stark contrast to traditional OpenFlow networks, where stateful processing happens mostly in control plane software.

Managing in-network data plane state, however, poses significant challenges. In an OpenFlow network, the centralized controller has a global view, and it plays an important role in state management tasks. In this context, a rich body of work has been developed centering around managing [15, 16, 43, 54, 59] and updating [17, 35, 44] network state in centralized SDN. In contrast, the counterparts of these algorithms in the setting of programmable data planes are much less well-understood. A strawman solution is to fall back to a centralized or switch-local controller, but this is often unattractive, as the control loop would significantly offset the benefit of keeping state in the data plane. Whereas data plane state can change per-packet at Tbps, control plane solutions operate at much larger timescales, and are therefore a poor fit for data plane state management.

In this paper, we consider the problem of *secure state migration*. Although data plane state management entails broader challenges, this primitive is an important building block for this general direction. Our goal is to develop a protocol for a switch to transfer its state to one or more destination switches effectively without involving a central controller, while providing strong integrity guarantees on the authenticity of the data and its sender. Many existing systems motivate the need for such a protocol: Contra [25] and Hula [29] require switches to propagate performance metrics across the network; FastFlex [57] requires switches to propagate attack detection results; and Swing State [37] develops program analysis techniques to identify state for migration, and sketches the first design for state migration. We design a new protocol that draws inspirations from live VM migration [12, 13, 32, 34, 47, 52, 55], while tackling practical challenges introduced by the P4 programming model.

Moreover, an important feature of our protocol is that it provides strong authenticity guarantees on the migrated state. Traditional control plane solutions can easily perform authentication on network state using cryptography, guaranteeing that received data has

SPIN '20, August 14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-8041-6/20/08…$15.00
https://doi.org/10.1145/3405669.3405822

indeed originated from the sender and that it has not been tampered with. However, when migrating state entirely in data plane, ensuring data authenticity is much harder. Programmable data planes have a restricted programming model, and they do not have built-in support for cryptography. Extending the programming model, whether using `extern` cryptographic hardware modules or using control plane CPUs, is possible but still insufficient: standard signature schemes such as RSA are inherently too slow to perform over high-speed data. Our proposal develops support by marrying two lines of work: a thread of work in search for lightweight cryptographic functions implementable in the P4 programming model [14, 23, 46], and another on authenticating data streams by amortizing signature overhead over a set of packets [41, 42].

The resulting protocol, P4Sync, is a principled design that can securely transfer data plane state with low overhead. Concretely, P4Sync first copies the main register arrays end-to-end using the switch packet generator with appropriate rate control. During this process, additional dirtied data is buffered in an auxiliary register array for aggregation, and is then copied in a similar fashion. To authenticate migration packets, P4Sync builds a hash chain by attaching each migration packet with a hash value that is computed using the content of this packet and the hash value of the last packet. The receiver can verify the integrity of this hash chain using per-packet data plane operations in the normal case. Finally, P4Sync relies on the control plane to sign the end of the chain using a public key signature, authenticating the entire migration sequence. We develop a prototype based on the P4 `bmv2` model, evaluate its effectiveness, and release the source code [6].

2 THE STATE MIGRATION PROBLEM

We give a more precise statement of the problem, and outline several design requirements for state migration.

2.1 Problem statement

The goal of state migration is to transfer a snapshot of stateful memory St from a source switch src to a destination switch dst. In programmable data planes, state St resides in a set of register arrays $R_1, \cdots, R_n$. Each register array, $R_i, i \in [1..n]$, has a pre-defined size s_i that is fixed at compilation time; and it can be indexed by an integer $j \in [1..n]$, where R_{ij} is the j-th register of R_i. We denote the largest array size as $s_{max} = \max\{s_1, \cdots, s_n\}$. State migration could be periodic events. One round of migration finishes when the destination switch has received St from the source switch. Across different rounds of migration, the source and destinations may also be different. Importantly, our definition is independent of the structure of the state (e.g., whether the registers are used as counters or part of a more advanced data structure, such as a count-min sketch or a bloom-filter); it is also independent from how state is used (e.g., whether a register is read-only or read-write, or registers may depend on each other).

Completeness. As a basic requirement, a migration protocol should guarantee that the state is completely copied from src to dst in each round. Achieving completeness in the migrated state would first require that the network provides reliable data transfer. This can be achieved by, e.g., prioritizing migration messages, adding FEC for error correction, and using acknowledgment and retransmission mechanisms on the migrated data. It also requires the migration protocol to sweep through the state efficiently in a systematic fashion.

Authentication. Furthermore, a receiver needs a way to check the integrity of the migrated data and to ensure that it comes from an authenticated sender. Symmetric cryptography, where the sender and receiver use a shared key for encryption and decryption, cannot provide non-repudiation guarantees in sender authentication. This is because any party holding the secret key could easily forge a message. In order to enable fault attribution—i.e., the ability to present cryptographic evidence that non-repudiably links a message to its sender—we need public key cryptography.

Overhead. Last but not least, the migration traffic should incur low overhead, because it shares the same network infrastructure with normal user traffic. Ideally, the migration protocol can dynamically control the rate of migration traffic, preventing potential congestion or microbursts.

2.2 State of the art & Limitations

Designing a migration protocol that achieves completeness, authentication, and low overhead presents unique challenges. As we discussed, only nascent work exists that consider managing data plane state [37, 51]. A notable project, Swing State [37], outlines a partial solution for state migration.

Swing state. The key insight of Swing State [37] is that register values can be piggybacked on network traffic, and that the migration traffic could be tunneled to the destination switches by header modifications in data plane. Concretely, consider migrating a register array R_i of size s_i. Swing State uses normal user traffic to piggyback values in the registers that packets may access. If a packet accesses register R_{ij}, then the switch clones this packet into two copies. One of these copies is sent to the original destination. The second copy is tagged with the value of R_{ij} as a special header, and sent to the migration destination.

Limitations. Swing State provides an interesting insight on leveraging network traffic for migration, but it also has notable limitations. *Completeness:* There is no guarantee that the migration will eventually transfer a complete snapshot of the sender state, because packets access the register array randomly and this may not cover the entire range. *Overhead:* As a result, the migration may take a long (or an indefinite amount of) time until all registers have been accessed by some packets. During this migration period, updates to the register array have to be repeatedly sent to the receiver. This causes high overhead. *Authentication:* Swing State also does not consider data or sender authentication.

3 THE P4SYNC MIGRATION

P4Sync builds upon the insight from Swing State—that we could migrate data plane state as packet headers. However, achieving the target properties in Section 2 requires a very different design. In this section, we describe the migration protocol, which is inspired by memory copy techniques developed in the context of live VM migration [12]; next section discusses authentication. Like Swing State, we assume that packet loss will be handled by out-of-band mechanisms, either using high priority for migration traffic or by retransmissions.

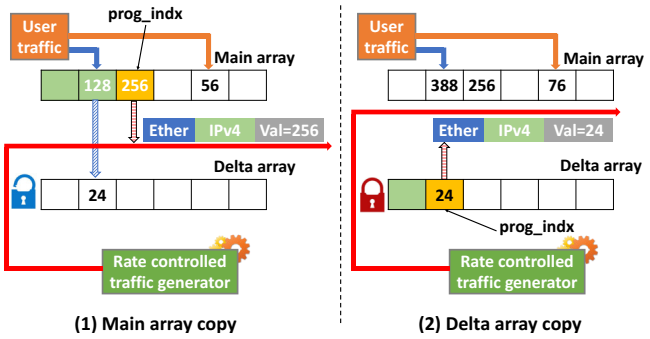

Figure 1: The key components of the P4Sync design.

3.1 Roadmap

The goal of our protocol is to transfer program state—register arrays in programmable data planes—from the source to the destination, while minimizing traffic overhead and the impact on normal user traffic. Figure 1 shows the key components of the protocol, which we detail next:

- *Copying the main array*: For each register array, P4Sync copies all its elements by sweeping through the registers, and attaching state on migration packets. Our data plane design maintains a progress index, which points to the next register to be copied, and advances it for each migration packet. If no packets update the registers during the copy, then the migration completes after the progress index points to the end of the array.
- *Incremental copy*: However, user traffic might update the main array as we perform the copy. If a register is updated after the first copy—i.e., the write location is smaller than the progress index, the register becomes "dirty". We record the updated delta into an auxiliary data structure—a delta array δ. The delta values are buffered for aggregation, and P4Sync sweeps through δ when the main copy finishes.
- *Point-in-time snapshot.* After P4Sync has transferred the last element in δ, it has migrated a *"point-in-time"* consistent snapshot [31], i.e., at this point in time, the sender has completely copied the state to the receiver. When copying δ, we pause any updates to δ; any subsequent updates to the main array will only be captured in the next migration round.
- *Rate control:* P4Sync uses the switch packet generator [7, 8, 60] to create migration packets, and controls the migration rate by adjusting the generator frequency. The rate control mitigates potential impact on normal traffic.

Unlike VM migration, which is typically one-shot, data plane protocols [25, 28, 57] usually migrate state periodically. In each round, the above protocol is invoked for efficient and complete migration. Migration from one source to multiple destinations can be handled using a similar protocol, with migration packets multicast to all destinations.

3.2 Achieving migration completeness

Programmable data planes, unlike multicore CPUs, do not have concurrency in terms of multithreading. Rather, they process one packet at a time, only allowing pipeline parallelism. However, modern switches have hardware packet generators [7, 8, 60] that can send traffic at tunable speeds. P4Sync leverages this source of concurrency to harness a thread of execution for state migration. It triggers the generator to send $\sum\{s_i\}, i \in [1..n]$ packets for the main copy, which migrates registers in $R_i, i \in [1..n]$. The `prog_indx` index guides the migration packets into the right slots to attach state, and the index is incremented by each migration packet. The same progress index can be reused across register arrays, by using an additional register to hold the ID of the current array under migration.

Compared to Swing State [37], which uses normal traffic for migration, leveraging the packet generator leads to several desirable properties—we can easily control the migration rate, and also avoid potential microbursts and congestion.

3.3 Minimizing migration overhead

Next, we need to handle registers that have been "dirtied" once again after they have been migrated in the main copy. The canonical approach in VM migration is to use a bitmap to keep track of dirty elements, and to iteratively migrate dirtied data until no more updates are made. However, this assumes that the program has a small *working set*, so fewer and fewer elements become dirty as we incrementally copy. But this is typically not the case for data plane state, since indexes to register arrays may be computed by CRC functions and produce random accesses. In other words, the dirty rate is very high and the working set could be as large as the entire memory.

P4Sync handles this by tracking dirty data using an auxiliary register array δ, whose length is s_{max}. If a write index is smaller than `prog_indx`, this register becomes dirty, and the delta is recorded in δ at the same index. Whether dirty or not, all writes to the main array are immediately effective and they are never stalled by migration. This δ array buffers and coalesces additional writes to reduce migration traffic. When P4Sync has migrated a main array, it proceeds to copy the delta array.

When copying δ, this array is not updated by additional writes any more. Rather, modifications to the main array will only be reflected in the next round of migration. P4Sync sweeps through δ using an index, and completes a *point-in-time consistent* [31] snapshot by the time δ has been migrated. Since δ is shared by all register arrays, we have a constraint that we can only migrate main arrays sequentially one after another. In the case where the switch memory has enough resources to hold a δ array for each main array, then P4Sync can migrate all arrays in parallel by attaching multiple register values to the same migration packet.

3.4 Controlling migration rate

Another critical aspect of state migration is rate control, similar as in VM migration [12]. This is because migration traffic may cause collateral damage to normal user traffic—additional traffic may produce congestion or packet drops, or they may cause microbursts in the switch buffers. P4Sync controls migration traffic rate by configuring the packet generator depending on the network condition. Suppose that we want the rate to be lower than R_m, and

the migration to finish within ϵ_t; and further assuming a propagation delay of d, a total state size of S_{reg}, and a migration packet size of S_{pkt}, then the packet generation frequency could be set to $R_m/S_{pkt} \leq P_{gen} \leq (\epsilon_t - d)/(S_{reg} + S_\delta)$. In addition, migration traffic could be load-balanced across multiple ports, e.g., using least-utilized switch ports at any given moment [21], to avoid creating network congestion.

4 THE P4SYNC AUTHENTICATION

Next, we describe how P4Sync performs efficient authentication to provide non-repudiation.

4.1 The threat model

We assume that the operator has deployed a correct migration protocol with a reasonable migration speed, and that the sender and the receiver switches are trusted, but other switches and links are not. The attacker's goal is to disrupt the functionalities of data plane applications by tampering with the migration protocol, either by modifying migration traffic maliciously or injecting/replaying spoofed packets via compromised switches or hosts. Under these assumptions, the attacker can launch at least three different types of attacks:

On-path tampering attacks. The adversary can attack the network by sniffing and tampering with the migration packets directly. For example, in Contra [25], a load balancing system, switches exchange migration packets that carry link utilization metrics to compute least-utilized routes. The attacker can cause load imbalance or even forwarding loops in the network by capturing and then modifying the link utilization data. She can also broadcast spoofed messages to say that a specific link in the network has very low utilization, causing large volume of traffic to be routed to that link; this would in effect launch a *link-flooding attack* [27, 50].

Off-path injection attacks. The attacker is also able to inject spoofed migration packets off-path. For instance, FastFlex [57] proposes a decentralized link-flooding defense in the data plane, where switches synchronize their reports on malicious flows with each other for network-wide defense. The attacker can inject spoofed synchronization packets, so that the switches misidentify legitimate flows to be malicious, causing them to be penalized or dropped. In addition, she can also launch "BGP poisoning" style [10] attacks to attract traffic to a link that she controls.

Replay attacks. An attacker can also capture legitimate migration packets from an authenticated switch, and re-injects them into the network later. It is well-known that replay attacks can be effective even when authentication mechanisms are in place [33]. As an example attack, consider Poise [28], which is an access control system that grants or denies access to network requests based on special "context" packets from clients. An attacker may be able to replay such packets to gain privileged permissions.

4.2 Efficient stream authentication

We now discuss how we authenticate a stream of migration packets efficiently. The starting point of our design is a protocol called EMSS [41], which considers the problem of performing efficient authentication over a stream of packets while avoiding the overhead of performing one public key signature per packet. Using this mechanism, P4Sync attaches a hash value to every migration packet. The hash value for the i-th packet is computed as $h(i) = h(P, h(i-1))$, where P is the current packet and $h(i-1)$ is the hash value of the previous packet. The receiver will therefore receive a chain of hash values, and can verify whether or not the hash chain is correctly constructed. Any incorrect hash values will cause the entire chain to be invalidated. At the end of the stream, the sender switch generates a special EOS (end of stream) packet that carries a public key signature of the final hash value. The receiver, using the public key of the sender, verifies whether or not this value authenticates the stream that has been received so far. We assume that the sender and receiver switch have exchanged their keys securely.

A special consideration is to handle transient states during migration and authentication. Since state transfer takes time to finish, there are two types of transient state. First, there will be a period of time during which the state has not been completely transferred. Therefore, the receiver state is only a subset of the sender state; this transient state is unavoidable. Second, there is a delay between the time when the destination receives the full state and the time when it receives the RSA signature. This is because the public key signature requires control plane involvement on a switch. On a modern CPU, it takes 1.5ms to generate a RSA-2048 signature [26]. If the receiver only starts using the data after receiving the RSA signature, this would create O(ms) additional latency. P4Sync allows receivers to start using unauthenticated data in this transient state, but quickly detects any tampering with a O(ms) delay. We believe that this is a reasonable tradeoff, because such an attack can only cause milliseconds of disruption. As soon as an attack is detected (e.g., RSA signature verification fails), the receiver raises an alarm to the operator and reports the attack.

In order to defend against replay attacks, P4Sync uses sequence numbers for the epochs and the migration packets per epoch. Concretely, time is divided into epochs, and each epoch has a unique ID *epoch*. Within each epoch, migrated state is associated with a sequence number *seq*, e.g., the register array ID and the progress index. An <*epoch, seq*> pair will uniquely identify the state sequence, and *epoch* and *seq* can only increase with time. Therefore, if an attacker replays an old packet of which the <*epoch, seq*> are outdated, the receiver will correctly reject it.

4.3 Handling packet loss and reordering

Upon packet reordering or loss, the receiver may not be able to authenticate a packet because the previous packet/hash is missing. P4Sync detects such cases by checking the <*epoch, seq*> pair, and use an advanced version of EMSS [41] to mitigate this problem. Specifically, P4Sync uses EMSS to construct multiple hash chains, where it attaches a packet with multiple hash values to build *skip list* like data structure. As Figure 2(a) shows, the sender computes $h(i_1) = h(P, h(i-2^0))$, $h(i_2) = h(P, h(i-2^1))$, ..., $h(i_n) = h(P, h(i-2^n))$, where $h(i-2^k)$ is the hash value of the 2^k-th packet before the current one. The receiver only needs to rely on one hash value to authenticate a received packet, and authentication only fails if all dependent hash chains are missing, which is unlikely.

To support this, the receiver uses an auxiliary register array of size $s_{max}$to maintain hash values of all previous packets. P4Sync

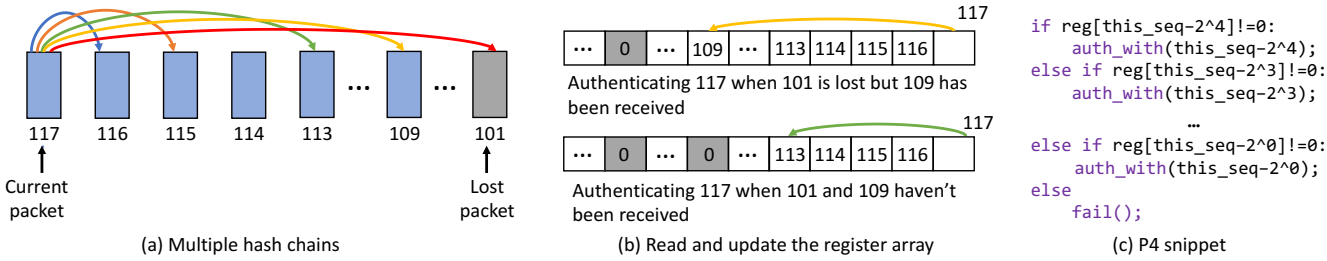

Figure 2: Data structures and program snippets for handling packet loss and reordering.

initializes all values in the register array to zero, and stores the hash values of authenticated packets in the positions indexed by their sequence numbers. Figures 2(b) and (c) show the process of authenticating a newly arrived packet. The packet carries multiple hash values that rely on different packets, ordered from the most recent to the least recent. P4Sync first tries to authenticate the current packet with the hash value that relies on the least recent packet, in case more recent packets have not yet arrived due to reordering. If the packet is authenticated successfully, P4Sync will cache its hash value in the register array. If the hash value of the previous packet is invalid (default: zero), which means the packet has not been received yet, P4Sync will try to use the hash value of the second least recent packet. For example, for packet 117, P4Sync will first try to use the hash value of packet 101 to authenticate it. If packet 101 has not been received, packet 117 is recirculated to index this array one more time, attempting to use 109 for authentication. In the worst-case scenario, none of the previous hash values have been received. In this case, P4Sync stores the hash value in the register array but marks it as unauthenticated. At the end of the migration, P4Sync uses the packet generator to sweep through this array end-to-end, and recompute hash chains for all such registers.

4.4 Discussions

Handling frequent migration. In some cases, the state migration happens very frequently. For instance, some systems [25, 29] may generate migration packets at RTT timescales. Therefore, the signature operations may still be invoked quite frequently. To handle this, we can perform further batching to use one signature per k *epochs*. This would further amortize the public key cryptography cost.

Handling denial of service. Our current design leaves denial-of-service attacks out of scope. Such an attacker can drop or modify migration packets so that the hash values are always rejected by the receiver. To mitigate this, we could consider *mimicry defenses* where migration traffic is hidden as normal user traffic. We plan to explore this in future work.

5 INITIAL VALIDATION

Setup. We have built a prototype of P4Sync for initial validation, which is written in P4 [5] for the migration logic and Python for traffic generation. It runs in Mininet `v2.2.2` with `bmv2` model. We adopt the same scenario with Swing State [37], where a source switch migrates a hash table that tracks packet counts of each flow in a network without packet loss and reordering. We also use Swing State as the baseline for comparison. The hash table is implemented using register arrays and uses CRC16 to calculate indexes with flow 4-tuples. We test different hash table sizes from 2^{10} to 2^{16}. We use SipHash [9] to build a hash chain and use RSA to sign and verify the last migration packet. SipHash [9] is a pseudorandom function (PRF) that can be used to derive keyed hashes, or HMACs, using a shared secret, and it can be implemented entirely in data plane; we borrow this insight from a previous project, SPINE [14]. To provide non-repudiation, the RSA operations run in the switch control plane. Our experiments mainly answer two questions: a) How efficiently can P4Sync migrate state? b) How effectively can P4Sync perform authentication?

Overhead. We first evaluate the migration overhead for different hash table sizes, by measuring the number of migration packets needed to completely migrate the hash table. If the register array size is n and a system uses N packets, we say that the overhead is N/n. As shown in Figure 3, P4Sync has an overhead of 2x, because it sweeps first through the main array and then the delta array. The baseline has much higher overhead (7.4x-14.9x). This is because user traffic produces random accesses to the register arrays, so it takes a long time to achieve complete migration (Section 2.2).

To further understand the overhead of the baseline, we further measure the completeness with different numbers of migration packets for the hash table with 2^{10} entries. As shown in Figure 4, the baseline already achieves 99.22% completeness when using 5x migration packets, but it uses another 3x to migrate the rest. This is expected, because as more entries are migrated, the probability that a normal data packet hits an entry that has not been migrated decreases gradually. Actually, this can be modeled as a coupons collector problem [3], and the expected number of packets needed to migrate the entire hash table can be computed as $n \times \sum_1^n 1/i$, where n is the register array size. Depending on the structure of the state, an incomplete state copy may not be usable, e.g., if registers depend on or refer to other registers that are missing.

In terms of memory overhead, we found that for a hash table of size 2^{16}, the memory overhead for P4Sync is 128 kB for the delta array. Since the network does not incur loss, the auxiliary registers for buffering hash values are not needed. If they are added to the program, they would incur a similar amount of overhead as the delta array, as each of them has size s_{max}. Overall, P4Sync incurs O(100kB) memory, only a fraction of O(10MB) memory in modern switches.

Authentication. To evaluate how effectively P4Sync performs header authentication, we simulate an attacker who launches two attacks: a) the attacker captures the migration packets and changes the carried state; and b) the attacker masquerades as the source

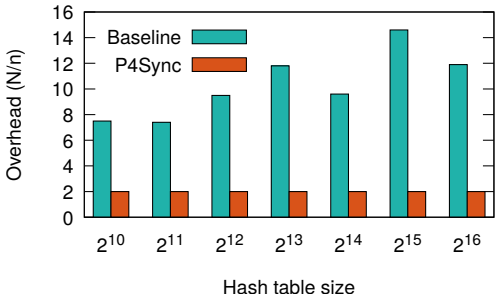

Figure 3: The overhead of completely migrating hash tables of different sizes.

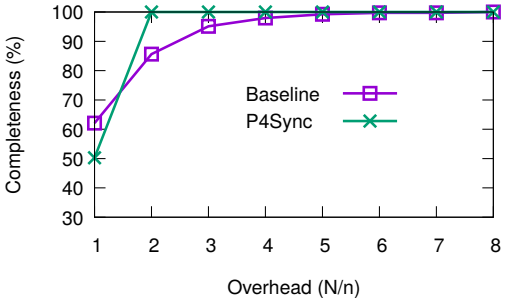

Figure 4: P4Sync has lower overhead and achieves completeness.

switch and sends spoofed migration packets to the destination. We repeat both attacks for 100 times and we find that P4Sync can always detect and report all attacks. The first attack is detected in the data plane when authenticating hash values; the second attack is detected by the control plane when verifying RSA signatures.

P4Sync provide non-repudiation guarantees by verifying the RSA signature of the last migration packet of each migration epoch. We measure the latency of generating and verifying a RSA signature in the control plane of a Wedge 100BF-32X Tofino switch, which has an eight-core Intel CPU at 1.60GHz. We find that it takes 2.15ms and 0.07ms on average to sign and verify a RSA-2048 signature. Therefore, the delay for authentication is under 3ms overall. To support more frequent migrations, P4Sync needs to be extended to perform one signature per multiple epochs.

6 RELATED WORK

State migration. State migration has been studied in traditional and SDN networks [20, 30, 36, 52]. For instance, VROOM migrates virtual routers across physical routers while minimizing disruption to user traffic. Swing State [37] and LOADER [51] sketch initial approaches to data plane migration. P4Sync proposes a principled migration protocol, and can provide authentication guarantees.

Crypto support in P4. A number of projects have considered cryptographic support in P4 [1, 2, 14, 22, 23, 46]. The most related work is SPINE [14], which uses SipHash [9] as a building block for header encryption.

Stream authentication. P4Sync builds upon previous proposals for amortizing signature cost [18, 40–42, 45, 53]. We adapt one of these protocols—EMSS [41]—for the data plane, and addresses several practical challenges, such as replay attacks, packet loss and reordering, and P4 support.

7 SUMMARY AND FUTURE WORK

We have described P4Sync, a state migration protocol that efficiently transfers switch state in the data plane, while leveraging signature amortization for authenticating the state and the sender. We hope that P4Sync will promote more discussions in the community on a) cryptographic support for data planes and b) state maintenance for P4 networks. Going forward, we are working to develop a hardware prototype and conduct a wider range of evaluations.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback; we also thank Kuo-Feng Hsu, Qiao Kang, and Yiming Qiu for their valuable comments on earlier drafts of this paper. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

REFERENCES

[1] Add crypto extern to behavioral-model. https://github.com/p4lang/behavioral-model/pull/834.
[2] AES encryption P4 implementation. https://github.com/chenxiaoqino/p4-projects/tree/master/AES.p4app.
[3] Coupon collector's problem. https://en.wikipedia.org/wiki/Coupon_collector%27s_problem.
[4] nplang. https://github.com/nplang.
[5] The P4 language repositories. https://github.com/p4lang.
[6] The P4Sync code repository. https://github.com/jiarong0907/P4Sync.
[7] Tofino: World's fastest P4-programmable Ethernet switch ASICs. https://www.barefootnetworks.com/products/brief-tofino/.
[8] Wedge 100bf-32x 100gbe data center switch. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.
[9] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: A fast short-input PRF. In *Proc. Indocrypt*, 2012.
[10] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with BGP. In *Proc. USENIX Security*, 2018.
[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM CCR*, 43(4):99–110, 2013.
[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proc. NSDI*, 2005.
[13] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. NSDI*, 2018.
[14] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.
[15] Roberto Doriguzzi-Corin, Elio Salvadori, Matteo Gerola, and Michele Santuari. An approach to exposing and sharing network services in software-defined networking. In *Proc. SOSR*, 2015.
[16] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proc. NSDI*, 2014.
[17] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. In *Proc. SIGCOMM*, 2014.
[18] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In *Proc. Crypto*, 1997.
[19] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of TCP. In *Proc. SOSR*, 2017.

[20] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, live migration of a software-defined network. In *Proc. SoCC*, 2014.

[21] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proc. SIGCOMM*, 2017.

[22] Frederik Hauser, Marco Häberle, Mark Schmidt, and Michael Menth. P4-IPsec: Implementation of IPsec gateways in P4 with SDN control for host-to-site scenarios. *arXiv preprint arXiv:1907.03593*, 2019.

[23] Frederik Hauser, Mark Schmidt, Marco Häberle, and Michael Menth. P4-MACsec: Dynamic topology monitoring and data layer protection with MACsec in P4-SDN. 2019. arXiv preprint arXiv:1904.07088.

[24] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *Proc. NSDI*, 2019.

[25] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[26] Ali Al Imem. Comparison and evaluation of digital signature schemes employed in ndn network. *arXiv preprint arXiv:1508.00184*, 2015.

[27] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. The Crossfire attack. In *Proc. S&P*, 2013.

[28] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[29] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[30] Eric Keller, Jennifer Rexford, and Jacobus E van der Merwe. Seamless BGP migration with router grafting. In *Proc. NSDI*, 2010.

[31] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proc. FAST*, 2013.

[32] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proc. SOSP*, 2017.

[33] Taeho Lee, Christos Pappas, Adrian Perrig, Virgil Gligor, and Yih-Chun Hu. The case for in-network replay suppression. In *Proc. Asia CCS*, 2017.

[34] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proc. HPDC*, 2009.

[35] Yujie Liu, Yong Li, Marco Canini, Yue Wang, and Jian Yuan. Scheduling multi-flow network updates in software-defined NFV systems. In *Proc. INFOCOM Workshops*, 2016.

[36] Samantha Lo, Mostafa Ammar, and Ellen Zegura. Design and analysis of schedules for virtual network migration. In *Proc. IFIP Networking*, 2013.

[37] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing State: Consistent updates for stateful and programmable data planes. In *Proc. SOSR*, 2017.

[38] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. NetHide: Secure and practical network topology obfuscation. In *Proc. USENIX Security*, 2018.

[39] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[40] Jung Min Park, Edwin KP Chong, and Howard Jay Siegel. Efficient multicast packet authentication using signature amortization. In *Proc. S&P*, 2002.

[41] Adrian Perrig, Ran Canetti, Dawn Song, and J. D. Tygar. Efficient authentication and signing of multicast streams over lossy channels. In *Proc. S&P*, 2000.

[42] Adrian Perrig, Ran Canetti, Dawn Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Proc. NDSS*, 2001.

[43] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *Proc. SIGCOMM*, 2013.

[44] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM CCR*, 42(4):323–334, 2012.

[45] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *Proc. CCS*, 1999.

[46] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. Cryptographic hashing in P4 data planes. In *Proc. ANCS*, 2019.

[47] Aidan Shribman and Benoit Hudzia. Pre-copy and post-copy VM live migration for memory intensive applications. In *Proc. Euro-Par*, 2012.

[48] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. USENIX ATC*, 2018.

[49] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proc. HotSDN*, 2013.

[50] Ahren Studer and Adrian Perrig. The Coremelt attack. In *Proc. ESORICS*, 2009.

[51] German Sviridov, Marco Bonola, Angelo Tulumello, Paolo Giaccone, Andrea Bianco, and Giuseppe Bianchi. Local decisions on replicated states (LOADER) in programmable data planes: programming abstraction and experimental evaluation. *arXiv preprint arXiv:2001.07670*, 2020.

[52] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus Van Der Merwe, and Jennifer Rexford. Virtual routers on the move: live router migration as a network-management primitive. *ACM SIGCOMM CCR*, 38(4):231–242, 2008.

[53] Chung Kei Wong and Simon S Lam. Digital signatures for flows and multicasts. In *Proc. ICNP*, 1998.

[54] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *Proc. NSDI*, 2018.

[55] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, Mazin S Yousif, et al. Black-box and gray-box strategies for virtual machine migration. In *Proc. NSDI*, 2007.

[56] Jiarong Xing, Qiao Kang, and Ang Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[57] Jiarong Xing, Wenqing Wu, and Ang Chen. Architecting programmable data plane defenses into the network with fastflex. In *Proc. HotNets*, 2019.

[58] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[59] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. PSI: Precise security instrumentation for enterprise networks. In *Proc. NDSS*, 2017.

[60] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. Hypertester: high-performance network testing driven by programmable switches. In *Proc. CoNEXT*, 2019.