P4EAD: Securing the In-band Control Channels on Commodity Programmable Switches

Archit Bhatnagar*†
National University of Singapore

Xin Zhe Khooi*‡
National University of Singapore

Cha Hwan Song
National University of Singapore

Mun Choon Chan
National University of Singapore

ABSTRACT

Conventionally, the control channel on network switches has always been out-of-band. With the emergence of high-performance systems built upon programmable switches, the out-of-band control channel has become the bottleneck. Thus, there is an emerging trend of implementing the control channel in the data path (i.e., in-band) on programmable switches to achieve high throughput and low-latency control actions. However, the use of in-band control channels comes with the risk of security vulnerabilities that have not been explored in prior literature. In this paper, we present `P4EAD`, a cryptographic primitive to secure the in-band control channels on programmable switches entirely in the data plane. This ensures the integrity, authenticity, and confidentiality of in-band control messages. We conduct micro-benchmarks on `P4EAD` and demonstrate its integration with an existing high-performance in-band control framework, showcasing minimal performance impact when securing the control channel.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Security and privacy** → *Symmetric cryptography and hash functions.*

KEYWORDS

P4, programmable switches, in-network, authenticated encryption, ASCON, in-band control channels

ACM Reference Format:
Archit Bhatnagar, Xin Zhe Khooi, Cha Hwan Song, and Mun Choon Chan. 2023. `P4EAD`: Securing the In-band Control Channels on Commodity Programmable Switches. In *Proceedings of the 6th European P4 Workshop (EuroP4 '23), December 8, 2023, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3630047.3630242

1 INTRODUCTION

The commoditization of programmable switches has ignited the "Cambrian explosion" of innovations in the network data plane. This has induced a paradigm shift in designing a new breed of high-performance mission-critical systems in the data plane, including stateful load balancers [21, 26, 33, 51], cloud gateways [36], application key-value store caches [23], 5G radio access network slicing mechanisms [13], 5G user-plane functions (UPF) [9, 30, 31, 40] etc.

While the packet processing capacity in the data plane has seen extensive growth over the years (e.g., up to 12.8 Tbps [17]), the I/O performance of the communication channel between the control plane and the data plane has largely fallen behind. Prior literature (DySO [41], AccelUPF [9]) have highlighted the significant difference in data plane forwarding rates (10^9 operations per second [16]) and control channel rates (10^4 to 10^5 operations per second [51]). The control plane thus incurs a non-negligible delay in reacting to ever-changing network dynamics and updating data plane states. This reaction delay significantly impacts system performance [14, 20, 41, 50, 52].

Generally, the bottleneck lies with the software agent [43] and drivers [41, 51] used conventionally by the control plane, known as *out-of-band control channels*, to communicate with the underlying switching ASIC over PCI-E channel. To address this issue, there is an emerging trend of the control plane shifting to using the high-speed data path to *directly* update the data plane states [9, 13, 41, 51], known as *in-band control channels*, bypassing the *slower* out-of-band control channel for greater system responsiveness and performance.

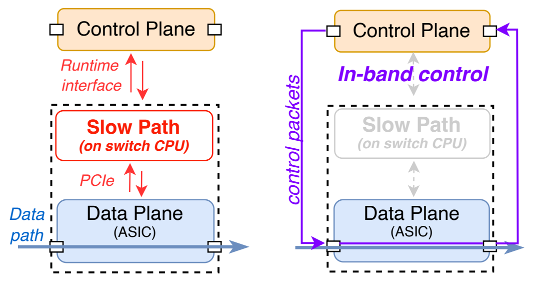
Insecure In-band Control Channels: The advent of *in-band control channels* prioritized performance over security. This departs from the conventional model of out-of-band control channels where various efforts have been dedicated to securing them [5, 8, 46]. As a result, the emerging use of *in-band control channels* opens up greater attack surfaces for adversaries to manipulate the system behavior through the injection of maliciously crafted, or spoofed control packets that affect the data plane states. To that end, *in-band control channels* require immediate attention to safeguard emerging high-performance networked systems.

Threat model: We consider an adversary within the network capable of intercepting, altering, and/or replaying *in-band* control packets. The adversary's aim is to hamper the system's performance and/or cause denial-of-service by manipulating the data plane states.

Securing the In-band Control Channels: Intuitively, this threat can be addressed by *securing* the in-band control channel to protect the authenticity and integrity of the corresponding in-band control packets. This necessitates the data plane to support the necessary cryptographic primitives. However, the restrictive pipeline programming model, limited hardware resources and instruction sets on commodity programmable switches make it challenging to

*Both authors contributed equally to the paper.
†Work done as part of the author's undergraduate thesis at BITS Pilani.
‡Corresponding author.

EuroP4 '23, December 8, 2023, Paris, France
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0446-8/23/12.
https://doi.org/10.1145/3630047.3630242



(a) Out-of-band. **(b) In-band.**

Figure 1: Overview on out-of-band/in-band control.

implement conventional cryptographic algorithms used in TLS [39] like AES-GCM [19] or ChaCha20-Poly1305 [34]. These algorithms involve complex operations such as modular arithmetic involving large numbers and an extensive number of loop iterations that are neither amenable nor practical to run on programmable switches.

To the best of our knowledge, this has not been investigated in the context of commodity programmable switches. Now that data plane behavior can be controlled through in-band control packets that are processed entirely in the data plane, there is a need to ensure that the forwarding states can only be modified by the "rightful" party. Thus, we identify `ASCON` [18], a lightweight cipher suite recently standardized by NIST [35] as a suitable candidate given its hardware-friendly design – that requires only binary operations like XOR, ROR, and AND. We present `P4EAD`[1], an implementation of the `ASCON-AEAD`[2] algorithm that runs entirely in the data plane to guarantee the authenticity and integrity of the control packets transmitted over the in-band control channel. In this paper, we detail the implementation challenges and solutions applied to implement `ASCON-AEAD` on the Intel Tofino switches. Our contributions are as follows:

1. `P4EAD`: An implementation of `ASCON-AEAD` on both the Intel Tofino and Tofino2 switches (§3) to enable secure in-band control channels. The code is publicly available at [4].
2. Extensive performance evaluations of `P4EAD` with different pipeline configurations, input length alongside their hardware resource trade-offs to act as a guideline for data plane developers (§4).
3. To demonstrate practicality, we integrate `P4EAD` to an in-network key value store [23] using the state-of-the-art in-band control channel framework [41] and observe negligible performance degradation (§5).

2 BACKGROUND AND RELATED WORK

In this section, we discuss the background and related work.

Control Channels on Programmable Switches: We begin by depicting the two types of control channels, namely out-of-band and in-band, that can be used with programmable switches in Fig. 1.

Out-of-band control channel: In out-of-band control channels, the control plane can either be *(i)* locally on the switch – talk to the OS kernel driver to communicate with the ASIC over PCI-E, or *(ii)* situated on a centralized server that interacts with multiple switches' local runtime interface [46] over secure TLS sessions (see Fig. 1a). It is well-studied in prior literature that the out-of-band control channel rates are slow [41, 43, 51] in updating/exporting data plane states. To that end, when compared to the data plane forwarding rates, the out-of-band control channel [41] lags far behind (10^9 versus 10^5) and incurs significant latency.

In-band control channel: With in-band control channels, the control plane essentially communicates with the data plane directly over the data path by sending dedicated *control packets* which contain the corresponding instructions and/or states (see Fig. 1b). This enables one to achieve a tighter control loop with higher throughput and low latency control operations in updating the data plane states. At the same time, the data plane states can also be managed by distributed control plane instances [41] (e.g., each instance manages a particular partition of the data plane states) for even greater throughput. Prior literature (DySO [41], AccelUPF [9]) has demonstrated that an in-band control channel approach can improve system performance up to two orders of magnitude.

Key differences: Performance aside, there are three key differences between out-of-band and in-band control channels, as follows: *(i) protocol used:* for out-of-band control channels, control actions are sent over TCP, which ensures reliable delivery, and they are usually secured using TLS [8, 46]. In the context of in-band control channels, there are no specific protocols used for it as control actions are carried over individual packets. To ensure reliable delivery, the application has to implement the necessary retransmission, and loss-detection mechanisms [41]; *(ii) control action packet size:* for in-band channels, the control packets must be within the maximum parseable length [12] of the programmable parser on the programmable switches. For cases where the control action is larger than the maximum length possible, then it is broken down into multiple packets. On the other hand, such limitations do not exist in out-of-band control channels; *(iii) data plane components that can be updated:* while in-band control channel approaches can update the data plane directly, they are limited to the registers (or stateful memory) only on existing programmable switches. On the other hand, its out-of-band counterpart can update both the match-action tables (MATs) and registers. To that end, existing works like DySO [41], Tiara [51], FSA [13], and AccelUPF [9] utilize multiple register arrays to construct MAT-equivalent functionalities for table lookup purposes, and high-speed in-band updates.

Authenticated Encryption: The NIST-standardized AES-GCM [19] and ChaCha20-Poly1305 [34] have been widely adopted and field-tested authenticated encryption schemes for secure communications, e.g., TLS [39]. They include associated data (AD), e.g., sequence numbers in the clear, as additional context bound to the ciphertext to prevent replay attacks [32]. They provide three key security properties, namely confidentiality, integrity, and authenticity. However, implementing them on commodity programmable switches is infeasible, if not impractical. For instance, AES-GCM involves the Galois Field (GF) multiplication process which has to be done bit by bit [45], resulting in an unacceptably slow performance. As for ChaCha20-Poly1305, it requires modular arithmetic performed on large numbers. On one hand, modular arithmetic is unavailable on commodity programmable switches. On

[1]pronounced as "paid".
[2]AEAD stands for Authenticated Encryption with Associated Data.

Round Constant Addition

$x_2 = x_2 \oplus c_r$

Substitution Layer

$x_0 = x_0 \oplus x_4$
$x_4 = x_4 \oplus x_3$
$x_2 = x_2 \oplus x_1$

Start of Keccak box

$t_0 = x_0 \oplus (\neg x_1 \wedge x_2)$
$t_1 = x_1 \oplus (\neg x_2 \wedge x_3)$
$t_2 = x_2 \oplus (\neg x_3 \wedge x_4)$
$t_3 = x_3 \oplus (\neg x_4 \wedge x_0)$
$t_4 = x_4 \oplus (\neg x_0 \wedge x_1)$

End of Keccak box

$t_1 = t_1 \oplus t_0$
$t_0 = t_0 \oplus t_4$
$t_3 = t_3 \oplus t_2$
$t_2 = \neg t_2$

Linear Diffusion Layer

$x_0 = t_0 \oplus (t_0 \ggg 19) \oplus (t_0 \ggg 28)$
$x_1 = t_1 \oplus (t_1 \ggg 61) \oplus (t_1 \ggg 39)$
$x_2 = t_2 \oplus (t_2 \ggg 1) \oplus (t_2 \ggg 6)$
$x_3 = t_3 \oplus (t_3 \ggg 10) \oplus (t_3 \ggg 17)$
$x_4 = t_4 \oplus (t_4 \ggg 7) \oplus (t_4 \ggg 41)$

$\oplus$ XOR $\wedge$ AND $\neg$ NOT $\ggg$ ROR

Figure 2: Overview of a `P-RND` in a `PERM`. Here, $x_0 ... x_4$ form the 320-bit `VECT` while $t_0 ... t_4$ are the intermediate state variables.

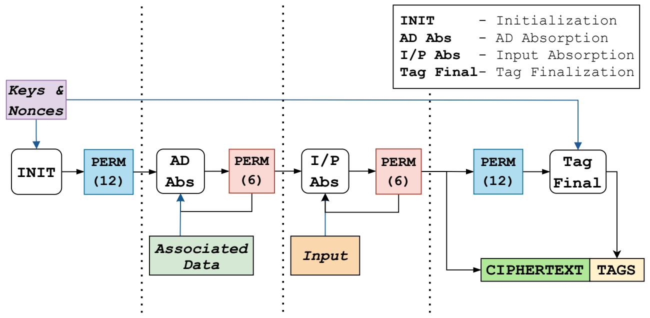


Figure 3: `ASCON-AEAD` authenticated encryption flow. Note that the decryption flow is symmetrical with encryption, with the exception that an additional tag verification action is done in the final stage.

the other hand, even if possible through lookup tables, it requires a non-negligible amount of SRAM in the data plane which makes it inherently expensive. ASCON-AEAD being part of the lightweight ASCON [18] cipher suite designed for resource-constrained systems, presents a much more amenable option for implementation on programmable switches over AES-GCM and ChaCha20-Poly1305, while offering similar security properties using fewer resources and being performant.

`ASCON` cipher suite: ASCON is a family of lightweight authenticated encryption and pseudorandom functions (PRF) which has been standardized for lightweight cryptography by NIST [6, 35] for resource-constrained systems. Particularly, ASCON relies on only simple operations (e.g., ADD, NOT, AND, ROR, and XOR), and the computations operate over a constant 320-bit state vector (VECT), making it memory efficient. At the core of ASCON, permutations (PERM) form the key building block, consisting of $n \times$ P-RNDs, denoted as PERM(n). A PERM is either PERM(6),PERM(8), or PERM(12). Each P-RND comprises three layers: *addition*, *substitution*, and *linear diffusion*, which operate on VECT iteratively. We depict the operations within a P-RND in Fig. 2.

In this paper, we focus on one of ASCON’s variants for authenticated encryption – ASCON-AEAD. Fig. 3 illustrates the high-level workflow of ASCON-AEAD. To perform authenticated encryption, there are four phases, namely initialization, associated data (AD) absorption, plain text absorption, and tag generation & finalization. In the init phase, the keys, nonces, and initialization vector are copied to VECT before performing a PERM(12). Then, the associated data (AD), e.g., IP header/ sequence number in the clear, is “absorbed” in 64-bit blocks and goes through PERM(6) sequentially. After AD absorption, an additional PERM(6) is performed. The same process is repeated for the plaintext to get the ciphertext. To get the tag, VECT is sent through another PERM(12) and is retrieved after XORing with the keys. The decryption flow is symmetrical to encryption, except that it comes with an additional tag verification step at the end for integrity checks.

Cryptographic algorithms on programmable switches: There have been several efforts on the implementation and adoption of cryptographic primitives on programmable switches like encryption schemes (e.g., P4-AES [15], ChaCha [49]), and secure keyed hash functions (SipHash [44, 47, 48]). However, existing schemes available cannot provide the three necessary security properties to ensure end-to-end secure communication, and thus P4EAD is orthogonal with the aforementioned. This also makes ASCON-AEAD the first cryptographic primitive in the data plane to support secure communication channels. While one can compose algorithms like P4-AES and SipHash to construct an authenticated encryption scheme, it is not proven secure [38] and requires more dedicated hardware resources. Further, the P4-AES approach is not scalable, as the number of sessions is strictly limited by the memory available to maintain the per-key precomputed lookup tables. In contrast, ASCON-AEAD requires little memory (see §3) to maintain the constants (i.e., 12 $\times$ 16-bit numbers) used for the P-RNDs. Given the low resource requirement of ASCON, it can be a viable option to be integrated into and co-exist with existing data plane programs to secure the in-band communication channels.

3 P4EAD

In this section, we present our implementation of P4EAD in P4 [11], designed for the Intel Tofino [16] switch. The discussion here also applies to the Intel Tofino2 [17]. The implementation of P4EAD is publicly available at [4].

3.1 Preliminaries

First, we discuss the preliminaries on programmable switches that relates to the P4EAD implementation. The switching pipeline consists of the Ingress and Egress blocks, which share common hardware resources [12] while remaining mutually exclusive. Thus, hardware resources are at a premium. A key challenge to implement P4EAD is that there is only a limited number of pipeline stages available (e.g., 12 [22] and 20 [3] on Tofino and Tofino2, respectively). Within each stage, multiple operations (e.g., binary/ arithmetic) can be performed on mutually exclusive variables by the ALUs. The resulting output of any computation at a particular stage will only be available to the next stage of the pipeline. This chain of dependency determines the number of stages required for any particular program [28]. If a program requires more stages than are available, the program cannot be compiled. Finally, if a computation cannot be completed within a pipeline pass, one common technique is to recirculate the packet back into the pipeline for another pass. This

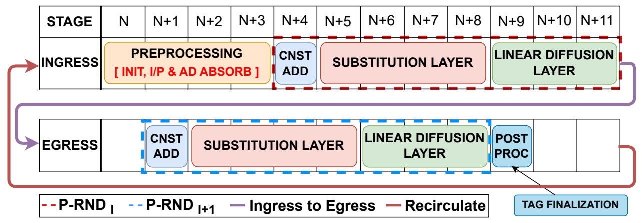


Figure 4: Stage layout of P4EAD on the Intel Tofino with a two P-RND per pipeline pass configuration.

is analogous to loops. However, packet recirculation consumes the available packet processing capacity, increases latency, and reduces throughput.

3.2 Implementation

PERM is a core component in ASCON-AEAD. Naively implementing PERM would result in a long, interwoven dependency chain that would not fit within the switching pipeline. To that end, we employ packet recirculations to do PERM over multiple pipeline passes – all phases would share the same P-RND implementation. We implement the P-RNDs in both the Ingress and Egress pipelines to increase the number of P-RNDs that can be performed within a single pipeline pass to reduce the number of recirculations required and its impact on throughput and latency.

Fig. 4 illustrates how ASCON-AEAD is implemented on the Intel Tofino. In this example, we perform two P-RNDs within a pipeline pass, one each at the Ingress and Egress blocks[3]. Here, we leverage the fact that PERM always has an even number of P-RNDs, i.e., PERM(6), PERM(8), or PERM(12), we expect PERM to terminate at the egress and proceed to the next phase at the ingress after the packet is recirculated. Thus, we perform preprocessing (i.e., initialization and absorption) at the Ingress and postprocessing (i.e., finalization) at the Egress. Note that the pre- and postprocessing blocks are bypassed when the packet is in the middle of a PERM.

Packet recirculation: Before recirculating packets, we add a special metadata header to the packet to store the current P-RND number and relevant states for stateful processing across recirculations. At the same time, we also save the original output port so that the packet can be forwarded accordingly at the end of the computation.

Preprocessing: The preprocessing block handles three kinds of packets: *(i)* newly received packets that needs to be encrypted/ decrypted, or *(ii)* recirculated packets in the middle of a PERM, or *(iii)* recirculated packets at the end of a PERM.

For *(i)*, we would initialize VECT, append it to the header stack, and then proceed to the initialization phase with the retrieved keys, nonces, and IV. The *keys* are always configured by the control plane onto the programmable switch over the secure out-of-band control channel. In the case of *(ii)*, the preprocessing block is bypassed. For *(iii)*, absorption is performed by XORing the next input block with VECT, before continuing with the next PERM of the next phase.

P-RND: Fig. 2 depicts the operations involved in a P-RND. To begin, each round starts with the addition layer, where a pre-defined

[3]More P-RNDs can be performed within a single pipeline by committing more hardware resources, if available.

Table 1: P4EAD's hardware resource consumption on the Intel Tofino and Tofino2 for the maxed-out configs.

Resource	tf1_2rnd.p4	tf2_4rnd.p4
SRAM	0.8%	0.8%
Hash Bits	13.2%	15.9%
Hash Dist. Unit	55.6%	66.7%
VLIW Ins.	6.5%	6.3%
PHV	29.3%	33.9%

constant is XORed with x_2 from VECT. We use a match-action table to implement a lookup table for the addition constants. This is implemented within a single pipeline stage.

It is then followed by the substitution layer which spans across 4 stages. Here, VECT is manipulated multiple times in a way that is non-trivial for the switch pipeline computation model. For example (refer to Fig. 2), take the operation $t_0 = x_0 \oplus (\neg x_1 \land x_2)$. This operation needs to be broken down into two parts. We have to compute $(\neg x_1 \land x_2)$ before getting the final result for t_0. This requires us to break down the operation into two separate stages and store the intermediate results as separate variables in the metadata.

The final layer is the linear diffusion layer which consists of XORs and ROR operations. We implement RORs using bit-slicing and concatenation [47]. Each operation involves two RORs and two XORs. Initially, we break down the operations in the diffusion layers into three parts by performing the RORs using the ALUs and PHVs naively and storing them using different intermediate variables during computation. This results in an extremely long dependency chain for the linear diffusion layer and the program would not compile. Instead, we exploit the hash engines within the pipeline stages to perform the RORs using identity hashing and then feed the three parts to the ALUs to perform XOR within a single stage. This allows us to implement the linear diffusion layer within 3 stages.

In total, our implementation of P-RND requires 8 stages.

Postprocessing: At the end of the final PERM tag finalization (and tag verification for decryption) is performed at the postprocessing block. After that the header containing VECT is stripped off before forwarding the resulting packet.

The P4EAD pipeline flow is summarized at appendix A.

Formula on number of recirculations. Let p and q be the length of the input and associated data in bytes, respectively. Then, we let r be the number of P-RNDs per pipeline pass. The total number of pipeline passes required, s, is given by:

$$s = \frac{30 + 6(\lfloor \frac{p}{8} \rfloor + \lfloor \frac{q}{8} \rfloor)}{r}$$

For example, for 8 byte input ($p = 8$), 4 byte AD length ($q = 4$) and 4 P-RNDs per pipeline pass using Tofino2 ($r = 4$), we require $\frac{30+6(0+1)}{4} = 9$ passes.

Hardware Resource Utilization We illustrate the H/W resource utilization for maxed-out configurations of P4EAD, i.e., 2 and 4 P-RNDs per pipeline pass for the Tofino and Tofino2, respectively in Table 1. Notably, our implementation heavily uses the hash distribution units (HDUs) for the linear diffusion layers in a P-RND. Here, each *individual* P-RND consumes 27.8% and 16.7% of the HDUs on the Tofino and Tofino2 respectively, for the maximal configuration (i.e., tf1_2rnd.p4 and tf2_4rnd.p4, respectively). As for the other H/W resources, they do not vary much except for the hash bits, which increase linearly with the number of P-RNDs.

4 EVALUATION

In this section, we benchmark the performance of P4EAD. First, we verify the correctness of our implementation against the reference implementation written in C [1] and the official test vectors. Next, we evaluate how the pipeline configurations (i.e., the number of P-RNDs per pipeline pass), and the input length (i.e., plaintext/ ciphertext) affect the throughput and latency. Finally, we also evaluate the scalability of our implementation by varying the number of recirculation ports dedicated for P4EAD. For brevity, we mainly focus our discussion on the results of the Intel Tofino2.

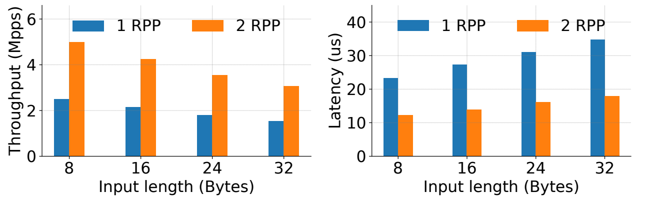
Evaluation Setup. Our setup consists of three Intel Tofino switches. We compile and run the different variants of P4EAD using the Intel P4 Studio v9.11.2 on a two-pipeline 3.2 Tbps Intel Tofino [16] and four-pipeline 12.8 Tbps Tofino2 [17] switch. For traffic generation, we make use of the packet generator on a separate Tofino switch. The switches are interconnected using 100 Gbps copper DACs.

Methodology. The length of associated data (AD) is fixed at 4 bytes, and we only vary the input (i.e., plaintext/ ciphertext) length. Unless otherwise stated, we use a single port for recirculation. To measure the best possible performance, we incrementally raise the packet generation rate until the point where there are no packet losses. We mainly highlight the encryption performance (with no packet loss) and omit decryption given that they are symmetrical. The experiments are automated and repeated 1000 times for consistency.

Interpreting the results. The presented results depict the maximum attainable performance of P4EAD under different configurations, i.e., P-RND per pipeline pass, and number of recirculation ports. Given the use of recirculation, the expected throughput will be lower than that of the recirculation port's speed. The use of recirculation is usually perceived "negatively" given its overhead. However, it is not the case if it is adequately planned, and performance is profiled while setting aside sufficient resources for crucial features like P4EAD. Moreover, it is worth noting that switches often have spare packet processing capacity [37], e.g., idle ports and/or under-utilized pipelines. Later (section §5), we showcase that the resulting performance still meets the requirements to perform fast in-band control channel updates.

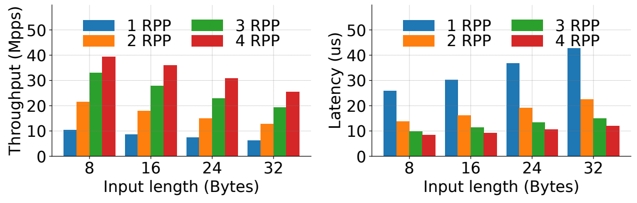
Impact of #P-RNDs per pipeline pass: We measure the throughput and latency for different pipeline configurations on both Intel Tofino and Tofino2 in Fig. 5 & Fig. 6 respectively. To ease the discussion, we first look at the case when the input length is 8 bytes as a similar trend applies to other input lengths regardless of the pipeline configuration. We observe that the throughput increases linearly from ~10 Mpps to ~40 Mpps when the number of P-RNDs per pipeline pass increases from 1 to 4 for the Intel Tofino2 (see Fig. 6a). As for the latency, we see a decrease from 25.6μs to 8.5μs (see Fig. 6b). The improved throughput and latency are attributed to the fewer recirculations needed with more P-RNDs per pipeline pass.

Impact of input length: Next, we evaluate the effect of different input lengths[4]. For brevity, we only highlight the results for the



(a) Throughput (b) Latency

Figure 5: Impact of the number of P-RNDs per pipeline pass (RPP) and input length for the Intel Tofino.



(a) Throughput (b) Latency

Figure 6: Impact of the number of P-RNDs per pipeline pass (RPP) and input length for the Intel Tofino2.

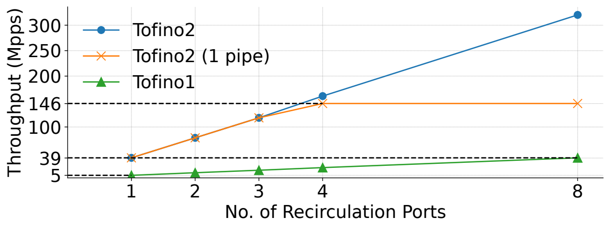
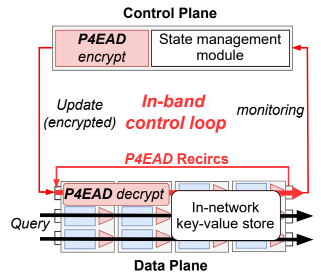


Figure 7: Impact of the number of #recirculation ports. The ports are 100 and 400 Gbps on Tofino and Tofino2.

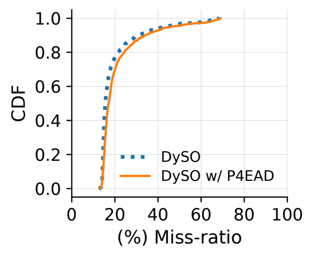
4 P-RND variant (the bars in red). From Fig. 6a, we observed that the throughput reduces from ~40 Mpps for an 8-byte input to ~25 Mpps when the input is 32 bytes. A reverse trend is seen for the latency in Fig. 6b from 8.5μs to 12μs. Recall that for every increase in 8 bytes, one additional PERM (6) is needed (see §2). This translates to decreased throughput and longer processing latency.

Impact of #recirculation ports: In Fig. 7, we demonstrate how P4EAD scales to support higher throughput by increasing the number of dedicated recirculation ports (up to 8) in Fig. 7 using the 2 P-RND and 4 P-RND variants for Tofino and Tofino2, respectively. As more recirculation ports are used, the throughput increases linearly up to ~320 Mpps for Tofino2. However, when the dedicated recirculation ports all belong to the same pipeline (analogous to a CPU core), the throughput tapers off at ~146 Mpps due to the pipeline being saturated at around 1.2 Bpps [3] with recirculated packets. This corresponds to the clock speed on the Tofino ASICs which is at around 1.2 GHz [3, 10]. This result indicates that dedicating an entire pipeline may not yield the best possible performance. Instead, the load should be shared across each pipeline respectively. Another insight is that the throughput for Tofino with 8 ports is approximately equal to the throughput for Tofino2 with only 1 port. By dedicating the entire Tofino pipeline with 16 ports (not shown in Fig. 7), the throughput saturates at ~66 Mpps.

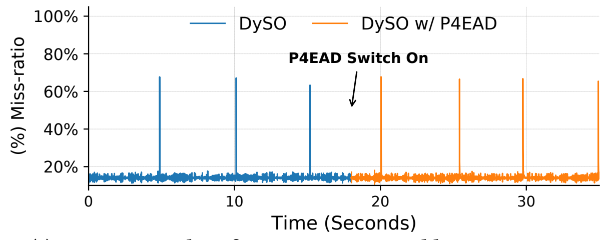
[4]This also applies to the associated data (AD).



(a) Evaluation setup for DySO-NetCache with P4EAD.



(b) CDF of DySO-NetCache miss ratio w/ and w/o P4EAD.



(c) Time series plot of miss-ratio. We enable P4EAD at t=18.

Figure 8: Impact of P4EAD on DySO-NetCache.

5 CASE STUDY: IN-NETWORK KV STORE

In this section, we present a case study by integrating P4EAD with an in-network key-value (KV) store that uses a high-speed in-band control framework and study the potential performance impact.

To meet aggressive latency and throughput objectives under highly skewed workloads, NetCache [23] caches hot entries on programmable switches to load balance the key-value store servers. As workloads can change rapidly in data centers, it is crucial that *stale* key entries can be replaced swiftly to minimize the miss ratio. However, as discussed in §2, the out-of-band control channel used to update the cached entries in the data plane has become the bottleneck. To address this, DySO [41] leverages the in-band control channel to accelerate the cache updates in NetCache (referred to as DySO-NetCache here onwards). As the cache entries are updated through the in-band control channel, it is exposed to the potential risks, where bogus cache updates are pushed to the data plane, thus causing denial-of-service.

To mitigate these risks, we secure DySO-NetCache with P4EAD to enable authenticated in-band updates to the cached entries in the data plane. We evaluate the performance impact of P4EAD by integrating it with DySO-NetCache's publicly available implementation [2] on the Intel Tofino switch.

Methodology. We configure P4EAD at 1 P-RND per pipeline pass, with the input length at 32 bytes to hold the key-value entries that are to be updated to the data plane. This configuration has a throughput and processing latency of ~1.5 Mpps and 34.9μs, respectively. We generate a key-value query stream with sudden workload changes, i.e., shifting $1K$ unpopular items to top-rank popularity every 5 seconds for 100 seconds. We use the same settings as the evaluation in [2, 41]. The setup is depicted in Fig. 8a.

Results. We see that despite P4EAD incurring additional latency overhead, the resulting performance is still comparable with the state-of-the-art [41]. Fig. 8b demonstrates the CDF of key miss-ratio, sampled every 1ms, with and without P4EAD. At the same time, Fig. 8c illustrates the time-series pattern of miss-ratio with P4EAD being enabled 18 seconds after the start. Both results demonstrate that P4EAD can be introduced into state-of-the-art in-band control channels without incurring a significant performance penalty on the application performance. Notwithstanding, P4EAD can be configured to support even higher in-band control channel update rates if needed (see §4) with an expected negligible impact on system performance.

6 DISCUSSION AND FUTURE WORK

In §5, we demonstrated P4EAD's practicality with an in-network key-value store system. We expect P4EAD can be introduced into other existing systems requiring in-band control channels without incurring performance degradations. For instance, in AccelUPF[9], the average latency to process an (in-band) PFCP packet on an Intel Tofino switch running AccelUPF was 35μs (see Table 1 in [9]). Assuming a minimal 1 RPP config for P4EAD with a latency around 72μs (2×36, see §4) for the longest input length, the total latency would be $(35+72) = 107\mu s$. This is still four times more performant than existing approaches. Besides, in the context of 5G fronthaul slicing in FSA [13], the data plane states need to be updated in a time-sensitive manner (i.e., every 1 ms). With a latency in the order of 10s of μs, we believe P4EAD will be feasible in such systems too. We plan to evaluate these systems with P4EAD in future work.

In the context of network inter-device communications, the significance of P4EAD becomes evident, particularly for enhancing security in emerging in-network applications. These applications encompass various forms of inter-device communication, such as switch-switch and host-switch interactions, wherein in-band signaling is employed to update data plane states. A notable example of this is observed in in-network load balancers [21, 26, 42], which generate probe packets to disseminate and evaluate network conditions, often involving vulnerable direct updates to data plane states. Furthermore, vulnerable in-network applications like network telemetry [7, 24], distributed state synchronization [29], DDoS defense mechanisms [27], and precise network-wide time synchronization [25] are also under consideration for integration with P4EAD as future work.

7 CONCLUSION

We present P4EAD, an implementation of ASCON-AEAD on commodity programmable switches for securing the emerging in-band control channels. We benchmark P4EAD on different configurations in terms of throughput and latency, as well as demonstrate its scalability. Finally, P4EAD incurs negligible overhead when integrated with existing applications using in-band control channels.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their invaluable feedback. We would also like to thank Raj Joshi, Satis Kumar Permal, and Yixi Chen for their insightful comments and suggestions on the paper. This work is supported by the NUS-NCS Joint Laboratory for Cyber Security, Singapore.

REFERENCES

[1] 2023. Ascon - Lightweight Authenticated Encryption & Hashing. https://github.com/ascon/ascon-c [Commit: ba61330].

[2] 2023. DySO P4. https://github.com/dyso-project/dyso_p4 [Commit: 4a594eb].

[3] 2023. Intel® Tofino 2 12.8 Tbps, 20 stage, 4 pipelines. https://www.intel.sg/content/www/xa/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html [Accessed: May 2023].

[4] 2023. P4EAD GitHub repository. https://github.com/NUS-CIR/tofino-ascon.

[5] AbdelRahman Abdou, Paul C van Oorschot, and Tao Wan. 2018. Comparative analysis of control plane security of SDN and conventional networks. *IEEE Communications Surveys & Tutorials* 20, 4 (2018).

[6] Jeffrey Avery, Bryson Fraelich, William Duran, Andrew Lee, Sullivan Agustin, Zane Mechalke, Birrer Maj Bobby, Sameul Dick, and Jordon Cochran. 2022. Analysis of Practical Application of Lightweight Cryptographic Algorithm ASCON. (2022).

[7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM 2021*.

[8] Kevin Benton, L Jean Camp, and Chris Small. 2013. OpenFlow vulnerability assessment. In *ACM SIGCOMM HotSDN 2013*.

[9] Abhik Bose, Shailendra Kirtikar, Shivaji Chirumamilla, Rinku Shah, and Mythili Vutukuru. 2022. AccelUPF: accelerating the 5G user plane using programmable hardware. In *ACM SOSR 2022*.

[10] Pat Bosshart. 2018. Programmable Forwarding Planes at Terabit/s Speeds. https://old.hotchips.org/hc30/2conf/2.02_Barefoot_Barefoot_Talk_at_HotChips_2018.pdf [Accessed: Mar. 2023].

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR* 44, 3 (2014).

[12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM CCR* 43, 4 (2013).

[13] Nishant Budhdev, Raj Joshi, Pravein Govindan Kannan, Mun Choon Chan, and Tulika Mitra. 2021. FSA: fronthaul slicing architecture for 5G using dataplane programmable switches. In *ACM MOBICOM 2021*.

[14] Huan Chen and Theophilus Benson. 2017. The Case for Making Tight Control Plane Latency Guarantees in SDN Switches. In *ACM SOSR 2017*.

[15] Xiaoqi Chen. 2020. Implementing AES encryption on programmable switches via scrambled lookup tables. In *ACM SIGCOMM SPIN 2020*.

[16] Intel Corporation. 2023. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html [Accessed: Mar. 2023].

[17] Intel Corporation. 2023. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html [Accessed: Mar. 2023].

[18] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2021. Ascon v1. 2: Lightweight authenticated encryption and hashing. *Journal of Cryptology* 34 (2021).

[19] Morris Dworkin(NIST). 2007. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. https://csrc.nist.gov/publications/detail/sp/800-38d/final. (2007).

[20] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Measuring Control Plane Latency in SDN-Enabled Switches. In *ACM SOSR 2015*.

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

[22] Intel. 2023. Intel® Tofino 6.4 Tbps, 4 pipelines. https://www.intel.sg/content/www/xa/en/products/sku/218643/intel-tofino-6-4-tbps-4-pipelines/specifications.html [Accessed: May 2023].

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP 2017*.

[24] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. 2021. Debugging Transient Faults in Data Centers using Synchronized Network-wide Packet Histories. In *USENIX NSDI 2021*.

[25] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. 2019. Precise Time-Synchronization in the Data-Plane Using Programmable Switching ASICs. In *ACM SOSR 2019*.

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

[27] Xin Zhe Khooi, Levente Csikor, Dinil Mon Divakaran, and Min Suk Kang. 2020. DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks. In *IEEE NetSoft 2020*.

[28] Xin Zhe Khooi, Levente Csikor, Jialin Li, Min Suk Kang, and Dinil Mon Divakaran. 2021. Revisiting Heavy-Hitter Detection on Commodity Programmable Switches. In *IEEE NetSoft 2021*.

[29] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. 2021. RedPlane: Enabling Fault-Tolerant Stateful in-Switch Applications. In *ACM SIGCOMM 2021*.

[30] Yunzhuo Liu, Hao Nie, Hui Cai, Bo Jiang, Pengyu Zhang, Yirui Liu, Yidong Yao, Xionglie Wei, Biao Lyu, Chenren Xu, Shunmin Zhu, and Xinbing Wang. 2023. X-Plane: A High-Throughput Large-Capacity 5G UPF. In *ACM MOBICOM 2023*.

[31] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay Thakur, Larry Peterson, Jennifer Rexford, and Oguz Sunay. 2021. A p4-based 5g user plane function. In *ACM SOSR 2021*.

[32] David McGrew. 2008. An Interface and Algorithms for Authenticated Encryption. RFC 5116.

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

[34] Yoav Nir and Adam Langley. 2018. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439.

[35] NIST. 2022. Lightweight Cryptography Standardization Process: NIST Selects Ascon. https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon [Accessed: May 2023].

[36] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. 2021. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM 2021*.

[37] Ting Qu, Raj Joshi, Mun Choon Chan, Ben Leong, Deke Guo, and Zhong Liu. 2019. SQR: In-network Packet Loss Recovery from Link Failures for Highly Reliable Datacenter Networks. In *IEEE ICNP 2019*.

[38] Phillip Rogaway. 2002. Authenticated-encryption with associated-data. In *ACM CCS 2002*.

[39] Yaron Sheffer, Peter Saint-Andre, and Thomas Fossati. 2022. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 9325.

[40] Suneet Kumar Singh, Christian Esteve Rothenberg, Jonatan Langlet, Andreas Kassler, Péter Vörös, Sándor Laki, and Gergely Pongrácz. 2022. Hybrid P4 Programmable Pipelines for 5G gNodeB and User Plane Functions. *IEEE Transactions on Mobile Computing* (2022).

[41] Cha Hwan Song, Xin Zhe Khooi, Dinil Mon Divakaran, and Mun Choon Chan. 2023. DySO: Enhancing application offload efficiency on programmable switches. *Computer Networks* 224 (2023), 109607.

[42] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. 2023. Network Load Balancing with In-network Reordering Support for RDMA. In *ACM SIGCOMM 2023*.

[43] Henning Stubbe, Sebastian Gallenmüller, Manuel Simon, Eric Hauser, Dominik Scholz, and Georg Carle. 2023. Keeping Up to Date With P4Runtime: An Analysis of Data Plane Updates on P4 Switches. In *IFIP Networking 2023*.

[44] Guangda Sun, Mingliang Jiang, Xin Zhe Khooi, Yunfan Li, and Jialin Li. 2023. NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering. In *ACM SIGCOMM 2023*.

[45] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *ACM ASPLOS 2022*.

[46] The P4.org API Working Group. [n. d.]. *P4Runtime Specification version 1.3*.

[47] Sophia Yoo and Xiaoqi Chen. 2021. Secure keyed hashing on programmable switches. In *ACM SIGCOMM SPIN 2021*.

[48] Sophia Yoo, Xiaoqi Chen, and Jennifer Rexford. 2024. SmartCookie: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes. In *USENIX Security 2024*.

[49] Yutaro Yoshinaka, Junji Takemasa, Yuki Koizumi, and Toru Hasegawa. 2022. On implementing ChaCha on a programmable switch. In *EuroP4 2022*.

[50] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive programmable switches. In *ACM SIGCOMM 2020*.

[51] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. 2022. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *USENIX NSDI 2022*.

[52] Annus Zulfiqar, Ben Pfaff, William Tu, Gianni Antichi, and Muhammad Shahbaz. 2023. The Slow Path Needs an Accelerator Too! *ACM SIGCOMM CCR* (April 2023).

Algorithm 1 `P4EAD pipeline flow`,

```
 1: procedure P4EAD_PIPELINE(pkt, curr_state, curr_rd, op_mode)
 2:     prnd_count ← 0
 3:     while prnd_count < RPP do
 4:         if curr_state == START then
 5:             curr_state ← INIT
 6:             pkt ← INIT(pkt)
 7:         else if curr_state == INIT then
 8:             if curr_rd == 12 then
 9:                 curr_state ← ABS_AD
10:                 pkt ← AD_ABS(pkt)
11:                 curr_rd ← 0
12:             end if
13:         else if curr_state == ABS_AD then
14:             if curr_rd == 6 then
15:                 curr_state ← ABS_IP
16:                 pkt ← IP_ABS(pkt)
17:                 curr_rd ← 0
18:             end if
19:         else if curr_state == ABS_IP then
20:             if curr_rd == 6 then
21:                 curr_state ← FINAL
22:                 curr_rd ← 0
23:             end if
24:         else if curr_state == FINAL then
25:             if curr_rd == 12 then
26:                 curr_state ← END
27:                 pkt ← TAG(pkt)
28:                 BREAK
29:             end if
30:         end if
31:         pkt ← P_RND(pkt)                          ▷ do one P-RND
32:         prnd_count ← prnd_count +1
33:         curr_rd ← curr_rd +1
34:     end while
35:     if curr_state == END then
36:         if op_mode == DECRYPT then
37:             valid_tag ← VERIFY_TAG(pkt)
38:             if ¬valid_tag then
39:                 DROP(pkt)
40:             end if
41:         end if
42:         pkt' ← pkt
43:         FORWARD(pkt')
44:     else
45:         RECIRCULATE(pkt)
46:     end if
47: end procedure
```

A `P4EAD` PIPELINE

We outline `P4EAD`'s pipeline flow in Alg. 1. Alg. 1 takes four inputs, i.e., pkt, curr_state, curr_rd and op_mode. Here, the current state (curr_state) can either be any of the following states: START, INIT, ABS_AD, ABS_IP, FINAL, and END. Similarly, the operation mode (op_mode) can be ENCRYPT or DECRYPT. In line 3, `RPP` denotes the number of `P-RND` per pipeline pass configuration of `P4EAD`, which can be 1 or 2 for Tofino and among 1,2,3, or 4 for Tofino2. For brevity, we assume that the AD and input absorption happens only once, i.e., the packet having a short AD and input, in Alg. 1. As larger input/ AD sizes, the absorption happens sequentially in chunks of 8 bytes.

When a packet is newly received, it is at the state START, and depending on the packet type, the op_mode is set. The recirculated packets which are in the middle of processing will have curr_state set to other than START. We keep track of the current `P-RND` using prnd_count and the current `ASCON` state using curr_state. A newly received packet's flow starts from the START state and moves onto INIT, which after `PERM`(12) moves onto AD absorption, and so on (as depicted in the flow in Fig. 3). The curr_rd parameter keeps track of the `P-RND` number for `PERM`(12) or `PERM`(6).

Finally, once the packet payload is absorbed and the tags are finalized, the END state is reached. If the op_mode is DECRYPT, then a tag verification happens before forwarding or proceeding with further packet processing logic with the final processed packet, pkt'.