Secure Keyed Hashing on Programmable Switches

Sophia Yoo
Princeton University
sophiayoo@princeton.edu

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

ABSTRACT

Cyclic Redundancy Check (CRC) is a computationally inexpensive function readily available in many high-speed networking devices, and thus it is used extensively as a hash function in many data-plane applications. However, CRC is not a true cryptographic hash function, and it leaves applications vulnerable to attack. While cryptographically secure hash functions exist, there is no fast and efficient implementation for such functions on high-speed programmable switches. In this paper, we introduce an implementation of a *secure keyed hash function* optimized for commodity programmable switches and capable of running entirely within the data plane. We implement HalfSipHash on the Barefoot Tofino switch by using dependency management schemes to conserve pipeline stages and slicing semantics for concise circular bit shift operations. We show that our efficient implementation performs 67 million, 90 million, 150 million, and 304 million hashes per second for 32-byte, 24-byte, 16-byte, and 8-byte input strings, respectively.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; **Network security**;

KEYWORDS

Data Plane, P4, Hash Function, SipHash, CRC32

ACM Reference Format:
Sophia Yoo and Xiaoqi Chen. 2021. Secure Keyed Hashing on Programmable Switches. In *ACM SIGCOMM Workshop on Secure Programmable network INfrastructure (SPIN '21), August 23, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3472873.3472881

1 INTRODUCTION

The emergence of high-speed programmable switches has enabled applications to run "within the network," improving security, privacy, performance, and reliability for network users. These applications often use hash functions to perform indexing in hash tables, generate short fingerprints for flow identifiers, and randomly sample a consistent subset of traffic. Many of these data-plane applications use CRC32 [21] to meet their hashing needs. Unfortunately, this introduces vulnerabilities to adversarial packets designed to exploit the weak hashing properties of CRC32.

SPIN '21, August 23, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8637-1/21/08…$15.00
https://doi.org/10.1145/3472873.3472881

1.1 Widespread Use of Insecure Hash Functions

CRC was designed to catch *accidental* bit flips and burst errors (i.e., a burst of consecutive 1s or 0s), which occur randomly in networks. However, CRC was not intended to catch *intentional* changes to the input message.

Thus, when it is used as a hash function in applications that require integrity, security vulnerabilities arise due to weak hashing properties.

CRC is linear, meaning that for any two messages A and B of the same length, we have

$$CRC(A) \oplus CRC(B) = CRC(A \oplus B) \quad (1)$$

Because of this property, an attacker can easily craft inputs to CRC in order to obtain their desired output values, i.e., construct hash collisions with legitimate inputs.

For example, recent work has shown that these inherent insecurities of CRC32 lead to WiFi hacking vulnerabilities in the IEEE 802.11 protocol [15]. In Table 1, we briefly survey several recent research works that use CRC32 in prototype data-plane applications, and note the security vulnerabilities they experience. We notice that hash functions are mainly used in three ways in these programs:

(1) **Indexing**: part of the CRC value of some header fields (such as flow ID) is used as an array index to implement hash tables. The adversary can cause collisions that force many table updates/evictions to go to the same index.
(2) **Fingerprint**: the CRC values are stored in the register array to represent longer flow IDs (or other keys). The adversary can cause collisions that let different flow IDs share the same fingerprint and inaccurately be grouped together.
(3) **Sampling**: CRC is used as a pseudo-random hash function, calculated over some packet header fields and compared with thresholds or ranges, to sample some subset of the input for measurement analytics. The adversary can construct traffic that is never sampled (bypassing monitoring) or always sampled (overloading the system).

Thus, when CRC is used in the data plane as a hash function, an attacker can induce many unintended behaviors, from causing simple slowdowns to exploiting severe security loopholes. Clearly, a safe and robust hash function is a critical building block for secure network applications. Yet, secure hash functions are often considered to be prohibitively expensive to compute, especially within the data plane of high-speed programmable switches. Without a better alternative, programmers continue to compromise security for lower computational cost and ease of implementation, by using CRC32 as their go-to hashing function.

Thus, we present the implementation of a secure keyed hash function for high-speed programmable switches running in the data plane. Our implementation will provide *speed, performance, and security* to fulfill the unmet secure hashing needs of network applications.

Application	Input Size	CRC usage			Vulnerability
		Indexing	Fingerprint	Sampling	
SilkRoad [19]	13/37 bytes		Yes		Target a single backend server
Cheetah LB [3]	13 bytes		Yes		
Jaqen [17]	12 bytes	Yes			Mislead penalty enforcement onto innocent flows
Poseidon [31]	4/13 bytes	Yes			
ConQuest [8]	13 bytes	Yes			
AROMA [4]	13/21 bytes	Yes		Yes	Bypass monitoring & Forge query output
BeauCoup [10]	2~18 bytes	Yes	Yes	Yes	
Cheetah [26]	4/8 bytes	Yes			
PRECISION [5]	8/13 bytes	Yes	Yes		Falsely trigger reports by collision
P4RTT [9]	16 bytes	Yes	Yes		
NetCache [14]	16 bytes	Yes			Reduce hit rate

Table 1: Prior works using CRC as hash functions in the data plane.

1.2 SipID: HalfSipHash in the Data Plane

We present *SipID* (Sip in the Data Plane), an implementation of HalfSipHash [2] that runs entirely in the data plane. HalfSipHash is a secure keyed hash function that is a good candidate for implementing on the pipeline-style computation model of programmable switches. However, there are several challenges inherent to implementing programs in the data plane. First, to enable line-rate packet processing, programmable switches using the PISA [6] pipeline architecture have strict constraints, such as a limited number of hardware pipeline stages. The switches also limit P4 programs to simple arithmetic operations and do not support operations such as division.

In this paper, we detail these challenges and our optimizations to overcome them and implement HalfSipHash on a Barefoot Tofino switch without compromising the recommended security parameters. We envision our effort to be the building block for many future networking applications running on programmable switches.

In summary, we make the following contributions:

- SipID: An implementation of a secure keyed hash function on the Barefoot Tofino programmable switch.
- An evaluation of optimized ingress-only versus ingress-plus-egress implementation variants.
- Performance results that show 150 million hashes per second for 16-byte input strings and over 300 million hashes per second for 8-byte input strings.
- The open-source code of our implementation[1] for community use in developing secure P4 applications.

We first overview SipHash and its 32-bit variant HalfSipHash in Section 2. Then, we describe challenges for implementing HalfSipHash in the data plane and our optimizations to overcome these challenges in Section 3. We present the results of our evaluation in Section 4, along with a discussion of our results in Section 5. Finally, we discuss related work in Section 6, and conclude the paper in Section 7.

2 SIPHASH BACKGROUND

In this section, we introduce the SipHash family of pseudorandom functions (PRFs) and the programmable-switch-friendly HalfSipHash variant.

[1]https://github.com/Princeton-Cabernet/p4-projects/tree/master/SipHash-tofino

SipHash is a family of keyed hash functions (i.e., PRFs) which performs a cryptographic hash of an input string using a secret key and generates a number that is indistinguishable from random. It was designed to achieve high speed and is optimized for short input strings [2], such as packet header fields that represent flow ID tuples. At a high level, the flow of SipHash is as follows: After taking an input string and a 128-bit secret key, it first initializes four internal 64-bit state variables. SipHash-c-d then performs c compression rounds followed by d finalization rounds on the input. These compression and finalization rounds are called SipRounds (Table 3) and are identical rounds, but with additional pre-processing and post-processing steps on specific rounds.

SipHash-c-d is believed to be cryptographically secure for all $c \geq 2$ and $d \geq 4$ [2]. SipHash is used by OpenDNS and in the standard libraries of Python, PHP, Rust, and Ruby, to more securely randomize hash table indexes [20, 29, 30]. By doing so, SipHash defends against hash collision attacks which force worst-case table lookup times. These prolonged lookup times result when a weak hash function maps an attacker's crafted inputs to the same hash index, resulting in collisions and CPU exhaustion from long searches at the collision index.

HalfSipHash is the 32-bit variant of SipHash and has the same core functionality as Chaskey[16], which is an efficient, permutation-based message authentication code (MAC) algorithm explicitly designed for speed on 32-bit microcontrollers. HalfSipHash uses the same SipRound operations as SipHash, but uses different shifting constants, takes a 64-bit key, and operates on 32-bit words. Four internal 32-bit words $v0$, $v1$, $v2$, and $v3$ are first initialized by XORing the upper and lower 32-bits of a 64-bit key with four 32-bit constants, as detailed in Table 2.

$v0 = k0 \oplus$ 0x70736575
$v1 = k1 \oplus$ 0x6e646f6d
$v2 = k0 \oplus$ 0x6e657261
$v3 = k1 \oplus$ 0x79746573

Table 2: HalfSipHash Initialization

1) $v0 \mathrel{+}= v1$	8) $v2 \mathrel{<<}= 16$
2) $v1 \mathrel{<<}= 5$	9) $v2 \mathrel{+}= v3$
3) $v1 \mathrel{\oplus}= v0$	10) $v3 \mathrel{<<}= 8$
4) $v0 \mathrel{<<}= 16$	11) $v3 \mathrel{\oplus}= v2$
5) $v2 \mathrel{+}= v1$	12) $v0 \mathrel{+}= v3$
6) $v1 \mathrel{<<}= 13$	13) $v3 \mathrel{<<}= 7$
7) $v1 \mathrel{\oplus}= v2$	14) $v3 \mathrel{\oplus}= v0$

Table 3: One HalfSipHash SipRound

After this, the first 32-bit word of the input message m_i is XORed with $v3$, the result is stored in $v3$, and then c compression SipRounds are executed. Each SipRound takes the four internal states $v0$ - $v3$ and iteratively updates these states using additions, XORs, and circular left shifts, as per Table 3. At the end of a SipRound, the internal states are passed to the next round for additional processing.

In HalfSipHash-2-4, each 32-bit word of the input string is processed by $c = 2$ SipRounds, and at the end of these rounds, the message word m_i is XORed with $v0$ and stored in $v0$. This process is repeated for each m_i in the input string. After the entire input has been processed, $v2$ is updated with a self XOR to the constant $0xff$, $d = 4$ finalization SipRounds are then executed, and finally the 32-bit output $[v0 \oplus v1 \oplus v2 \oplus v3]$ is returned.

For example, a 16-byte input will be split into four words, resulting in a total of 12 SipRounds: four groups of two compression SipRounds followed by four finalization SipRounds (2+2+2+2+4).

HalfSipHash-c-d is expected to provide maximum security for any keyed hash function with the same key size and output size, given compression and finalization rounds with $c \geq 2$ and $d \geq 4$, respectively [1]. By design, HalfSipHash performs most optimally for short input messages. We note that many programmable switch applications are interested in hashing packet header fields (not full packet payloads), and thus would benefit greatly from HalfSipHash's excellent security and speed on short inputs. Additionally, HalfSipHash in the data plane would be good for applications that might only need to hash a small fraction of packets, such as when a connection is first established. We also note that first-generation P4 programmable switches store variables in 32-bit containers natively. Since HalfSipHash operates on 32-bit words, this makes it an attractive choice for data-plane applications that require *security* and *high performance*.

3 SIPID IMPLEMENTATION

We now present our implementation of HalfSipHash on P4 programmable switches. We focus on HalfSipHash-2-4 due to its good security-versus-speed tradeoff, but our design can easily be used to support other HalfSipHash-c-d variations.

3.1 Limited Number of Pipeline Stages

Programmable switches have a limited number of pipeline stages available. Within each pipeline stage, the program can perform several arithmetic operations (such as add, xor, shift, etc.) concurrently on different variables. However, the output results are not available for use until the next pipeline stage, so other operations depending on these results must wait to be processed in this next stage.

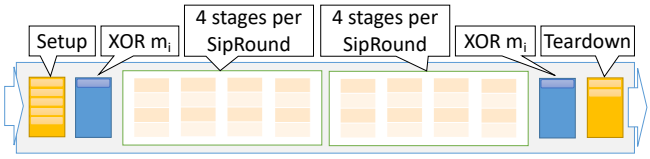

Figure 1: We run two SipRounds per switch pipeline pass, with each SipRound costing four stages.

HalfSipHash iteratively updates four internal state variables in repeated SipRounds, which creates a long, interwoven dependency chain. Naively implementing SipRound per the algorithm in Table 3 would thus cost too many stages or even consume the entire pipeline. To fit a SipRound within the switch processing pipeline, we help the compiler recognize dependencies by making these dependencies explicit with variable renaming and by grouping operations based on their dependencies. We also use packet recirculation and combined ingress + egress pipeline stages to further optimize HalfSipHash for the data plane.

Variable Renaming. We use a separate set of variables $a0$ - $a3$ and $b0$ - $b3$ in addition to the internal state variables $v0$ - $v3$ so that all operations will have distinct input and output variable names. This makes the dependency relationships between the different operations more clear, and also makes it easier for us to arrange the operations into separate actions for the P4 compiler.

Dependency Grouping. We choose to group the operations in one SipRound into four separate stages, such that each of the four internal state variables are written to once in each stage and never accessed in the same stage again after being written. We present dependency groupings for a full SipRound below.

Stage 1	Stage 2	Stage 3	Stage 4
$a0 = v0 + v1$	$b1 = a1 \oplus a0$	$a2 = b2 + b1$	$v1 = a1 \oplus a2$
$a2 = v2 + v3$	$b3 = a3 \oplus a2$	$a0 = b0 + b3$	$v3 = a3 \oplus a0$
$a1 = v1 << 5$	$b0 = a0 << 16$	$a1 = b1 << 13$	$v2 = a2 << 16$
$a3 = v3 << 8$	$b2 = a2$	$a3 = b3 << 7$	$v0 = a0$

Table 4: SipRound Operations Grouped by Dependencies into Four Pipeline Stages

Using these dependency groupings, we can implement k SipRounds in $k \times 4$ stages, with an additional s stages for setup and teardown. If the total pipeline length is S stages, we can perform $k = \lfloor (S-s)/4 \rfloor$ SipRounds per pipeline pass. Then, with a given input string size of M bytes, we require the number of passes given by p:

$$p = \left(\frac{M}{4} \times c + d \right) / k \tag{2}$$

Our implementation performs $k = 2$ SipRounds per pipeline pass, as illustrated in Figure 1. With $M = 16$ bytes for an input string and the recommended security parameters $c = 2$, $d = 4$, we need a total of $p = (\frac{16}{4} * 2 + 4)/(2) = 6$ pipeline passes to complete all $(2 + 2 + 2 + 2) + 4 = 12$ compression and finalization SipRounds.

Packet Recirculation. Since we cannot complete all the SipRounds required by HalfSipHash within one pipeline pass (for a 16-byte input, 6 pipeline passes are needed), we utilize the programmable switch's packet *recirculation* feature, which sends the packet to

special recirculation ports that simply bounce the packet back to the start of the ingress pipeline. Before a packet is recirculated, we add a special metadata header to the packet to store internal states and the current round number. We also save the original output port in this metadata header, so the packet can be routed to the output port at the end of the HalfSipHash computation. We use match-action tables with exact match rules to define high-level logic per recirculation pass and perform the correct number of compression and finalization SipRounds.

Ingress+Egress Pipeline. To further optimize SipID, we design a variation of our implementation that uses both the ingress and the egress processing pipelines, reducing the number of recirculations by a factor of two. More specifically, given $w = M/4$ words of an M-byte input string, HalfSipHash-2-4 needs to run $2w$ compression rounds and 4 finalization rounds. Therefore, for our ingress-only variant, we need to run $2w + 4$ rounds in total and require $r = (2w + 4)/2 - 1 = w + 1$ recirculations. When we run calculations in both the ingress and egress pipelines, we reduce the number of recirculations needed to $r = (2w + 4)/4 - 1 = w/2$.

For example, with a 16-byte input processed by our ingress-only variant, $w = 16/4 = 4$ and we require $r = w + 1 = 5$ recirculations. Given the same input string size, by using our ingress+egress variant, we reduce r to be $r = w/2 = 2$ recirculations per HalfSipHash calculation.

3.2 Limited Arithmetic Operations

In addition to being constrained to a limited number of pipeline stages, programmable switches are also restricted to specific types of arithmetic operations. In practice, because of these arithmetic limitations, the SipRound groupings proposed in Table 4 cannot be trivially implemented in the switch within strictly four stages.

Specifically, circular bit shifts are the arithmetic operation that consume too many stages. Our programmable switch does not natively support circular shifts using its Arithmetic Logic Units (ALU) and requires separate intermediate operations. A naive implementation of a circular left shift of n bits would need to first calculate the left and right half of the output using two separate shifts (left shift n bits and right shift $32 - n$ bits), and then combine the two intermediate results via bitwise OR:

$$b = (a << n) \;||\; (a >> (32 - n)) \tag{3}$$

Because the arithmetic results are not available within the same pipeline stage for immediate use, the bitwise OR would need to happen in the next pipeline stage, thus consuming two stages for a single circular bit shift. Each SipRound requires six circular left shifts, and using this approach, the number of pipeline stages required quickly spikes so that a full SipRound can no longer reasonably fit into a single pipeline pass.

Slicing for Circular Bit Shifts. Instead of using this two-stage approach, we notice that the P4 language supports slicing semantics to extract a subset of bits from a variable. We can thus calculate the same circular left shift by slicing and concatenating bits of the variable as follows, where n is the number of bits to be shifted and ++ is concatenation:

$$b = a[(31 - n) : 0] \;++\; a[31 : (32 - n)] \tag{4}$$

With this implementation, a circular shift can be calculated within a single pipeline stage, without the need to use extra intermediate variables. Thus, we were able to reduce the number of stages required for a circular left shift, and ensure that all six circular left shifts as well as the other addition and XOR operations in a SipRound can complete within four hardware pipeline stages as proposed in our design previously.

4 EVALUATION

In this section, we present performance evaluations for our HalfSipHash implementation in the data plane. We first checked the correctness of our implementation against the reference implementation written in C [1]. We tested both the ingress-only and ingress+egress variations of our implementation using different input lengths.

Setup. We compile and run each of the variants of our P4 HalfSipHash implementation on a Barefoot Tofino Wedge32X-100BF programmable switch, with 32 physical ports each operating at 100Gbps. The switch is connected to two servers, each with two 2.2GHz Intel Xeon 4114 10-core CPUs across two NUMA nodes and 96GB memory. Each server also has a Mellanox ConnextX-5 NIC with two 100Gbps ports.

On the servers, we run DPDK 19.11 and Pktgen 19.12 to generate and send minimum-sized packets (42 bytes header + hash input) to the programmable switch, while also observing the transmitted/received packets per second in real time. When the sending rate exceeds the maximum processing speed of the P4 program, packet drops occur and the receiving rate drops below the sending rate.

Each port on our programmable switch is limited to 100Gbps. To test our P4 program at above 100Gbps line rate (maximum 148.8 million packets per second), we use multiple links from two servers to simultaneously send packets to the switch. We also configure the switch to distribute its final outgoing packets randomly between two or four ports. This allows us to measure sending and receiving rates as high as 400Gbps (595.2 million packets per second).

Ingress + Egress Pipeline. As discussed in Section 3, our initial implementation of HalfSipHash with a 16-byte input uses only the ingress pipeline, thus requiring 5 packet recirculations in total to finish 12 SipRounds (6 pipeline passes). This severely limits the performance of the implementation: by default, our Tofino switch reserves 200Gbps of recirculation throughput, that must be shared by all the recirculated packets. To achieve higher performance, we need to either reserve more physical ports as recirculation ports or reduce the number of recirculations per packet.

By also performing HalfSipHash rounds in the egress pipeline, we can halve the number of pipeline passes needed, and use significantly fewer recirculations per packet. We run a benchmark experiment using 16-byte inputs to quantify the performance improvement. In Figure 2, we compare the packet processing rate of two variants of our HalfSipHash implementation: a baseline variant that uses only the ingress pipeline and an optimized variant using both ingress and egress pipelines for processing. As we can see, the optimized variant using both ingress and egress pipelines more than doubles the hash rate and also increases the maximum hash rate by 3x. This is because we reduced five recirculations (six passes total) down to two recirculations (three ingress+egress passes) per packet.

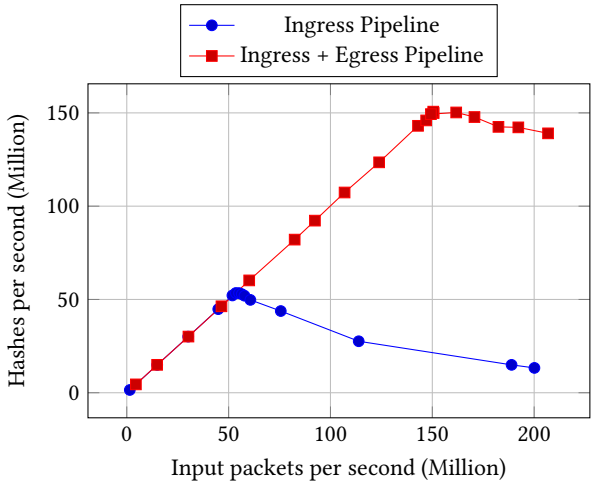

Figure 2: By using both ingress and egress pipelines, we increased our implementation's maximum hash rate by 3x.

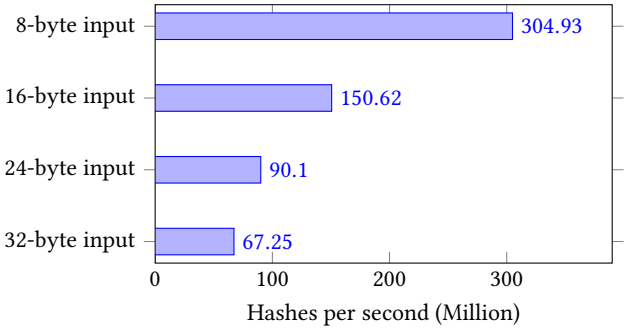

Figure 3: Hash rate of HalfSipHash on programmable switch under different input length.

	Ingress-Only	Ingress+Egress
TCAM	0.0%	0.0%
SRAM	0.3%	0.6%
Instructions	8.3%	9.4%
Hash Units	33.3%	66.7%
PHV	14.2%	27.8%

Table 5: Hardware resource utilization of our HalfSipHash implementation.

Effect of Input Length. Since the SipHash family of pseudorandom functions processes bytes of an input string sequentially, different input lengths require different numbers of hashing rounds. In Figure 3, we compare the performance of our optimized ingress+egress variant of HalfSipHash, given different input lengths varying from 8 bytes to 32 bytes.

As we can see in Figure 3, the 8-byte input variant (which uses only 1 recirculation) achieves a performance of 304.93 million hashes per second. This is near the maximum theoretical result for 1-recirculation programs, given that dividing the 200Gbps (200×2^{30} bits per second) reserved recirculation throughput on our switch by the minimum Ethernet frame size (64 bytes frame plus 20 bytes gap) equals 319.6 million packets per second.

We observe that the hash rate drops for longer messages. However, for many networking applications, hashing short input strings is sufficient. For example, many typical applications hash the 13-byte flow ID 5-tuple (source/destination IP pairs, source/destination port pairs, and protocol), and this can easily be fed into HalfSipHash as a 16-byte input with constant padding. Thus, our implementation already meets the needs of many of the applications shown in Table 1, allowing these applications to securely calculate hash values at a rate of 150 million flow IDs per second. In addition, other applications such as key-value store caching might use shorter 12-byte or 8-byte inputs, achieving even faster hash rates.

To put the experiment results into context, we also run a simple benchmark: one modern CPU core (Intel Xeon 4114 @2.20GHz) can run the reference C implementation of HalfSipHash-2-4 with 16-byte inputs at 27 million hashes per second (and can run the 64-bit variant SipHash at 33 Mh/s). Prior benchmarks have also shown a performance of approximately 33 Mh/s for SipHash-2-4 with 16-byte inputs on one CPU core [22], and comparable performance on NetFPGA SUME (14Mh/s) [22] or Netronome NFP-4000 SmartNIC (estimated 33-44Mh/s based on [22]).

Resource Utilization. Our HalfSipHash implementation uses only minimal hardware resources on the switch, leaving plenty of space for running other operations and advanced applications that are complemented by a secure hashing functionality. We present the hardware resource utilization of our implementation in Table 5.

Our most complex P4 program, the optimized variant using both ingress and egress pipelines, only uses 27.8% of packet header vector (PHV) for storing variables, 9.4% of available instruction words, and negligible amounts of ternary content-addressable memory (TCAM) and static RAM (SRAM). It uses 66.7% of available hash units, purposed towards efficiently implementing cyclic bit shifts.

We expect that other programs co-existing on a switch with our HalfSipHash implementation can benefit from secure hashing, and thus be less dependent on using hash units for the more traditional purpose of calculating CRC. However, there is more than one way to implement cyclic bit shifts on the Tofino switch. If the HalfSipHash calculation needs to coexist with other application logic that heavily uses traditional CRC hash functions, we can free up hash units and use the stateful ALU to implement the same cyclic bit shifts. We have tested that the stateful ALU-based alternative implementation still achieves the same hashing performance.

5 DISCUSSION

In the future, we plan to encapsulate SipID into more simple language primitives in P4, making it easier for data-plane application developers to integrate secure keyed hash functions into their P4 programs. We would also like to test our implementation on other programmable switches on the market and ensure our implementation is target-independent.

An actual P4 application using SipID might require multiple pipeline passes for its own application logic. The HalfSipHash calculation may happen simultaneously with the application logic not

requiring the hash value, reducing the total number of recirculations needed. However, we also note that routing decisions can only be made in the Ingress pipeline. If the application depends on the hash values for routing decisions and the hash values are only available at the end of the egress pipeline pass, the application needs one extra recirculation before making routing decisions.

Although SipID allows calculating up to 300 million hashes per second, it is still an order of magnitude slower than the Tofino switch's total packet processing capacity. Therefore, SipID is more useful for applications that only calculate hash values for a subset of packets, for example at the beginning of every flow connection. Calculating HalfSipHash for every packet may be too costly, and a potential workaround is to cache recent flow IDs and their hash values in stateful memory registers and only perform full hash calculations upon cache misses. We leave this as future work.

We believe future programmable switch hardware should natively support widely-used cryptography primitives, similar to modern CPUs supporting the AES-NI instruction set, making it easy for various data-plane application to use secure hashing and encryption. The switching pipeline could include fixed-function circuits dedicated for computing a secure hash function (or a step of it), completing the computation within one or a few stages. Until then, SipID can act as a stop-gap solution. Based on our experience implementing HalfSipHash and AES [7] on programmable switches, implementing cryptographically secure algorithms requires many pipeline stages, since these algorithms usually repeat a construction multiple times. Thus, an alternative to adding a fixed-function cryptography circuit is to add very deep pipelines or optimize the switching chip I/O for multiple pipeline passes (recirculations). This option is more future-proof and will survive the evolution of hashing and encryption standards.

6 RELATED WORK

Hash Functions. Network and system applications use fast (but insecure) hash functions such as FNV1, jhash, fasthash, xxhash, and murmurhash etc. extensively. For some system applications such as database sharding, fast and insecure hash functions are oftentimes sufficient, as the input is not adversarial. However, for network applications processing potentially adversarial traffic, stronger hashing guarantees are critical. This is because adversaries can remotely send packets that force weak hash functions to lose randomness and easily allow hashing collisions.

Cryptographic hash functions such as MD5, SHA-1, SHA-3, and BLAKE etc. were designed to provide security guarantees. In 2005, Wang and Yu [28] broke MD5 by providing a method to construct collisions, and in recent years researchers have made these attacks more efficient and practical [18, 23, 24]. Thus, applications should avoid using MD5 and migrate to stronger hash functions like SHA-3.

Even with a perfectly secure random hash function, hash collisions occur naturally [25]. Increasing the output length of a hash function is one way to trivially boost security by increasing the output search space and reducing the success probability of an attacker. The authors of HalfSipHash proposed a method to double the output of the function to 64 bits (from 32 bits) by running one more finalization SipRound over the internal states at the end of the function. The outputs of the last two finalization rounds are combined into a 64-bit final output. We can implement this method to boost hashing security for data-plane applications as well. However, we note long outputs are unnecessary for applications that need only a few output bits, for example with hash table indexing.

Data Plane Cryptography. Scholz et al. [22] proposed to add new cryptographic hash function primitives to the P4 language (using the *externs* feature of P4 specifications). The authors introduced prototype implementations of cryptographic hash functions on CPU backends (the t4p4s compiler), an NFP-4000 SmartNIC, and a NetFPGA Sume board. Among their hash function implementations, which include Poly1305-AES, BLAKE2b, HMAC-SHA512, and HMAC-SHA256, they identify SipHash as the hash function with the best performance and lowest latency on their target t4p4s CPU. Our implementation on programmable switches with PISA-based pipeline architecture complements this work.

Similarly, Hauser et al. [11, 12] proposed P4 extern semantics to implement MACSec and IPSec (network traffic encryption protocols) on P4 programs running on CPU backends. The work did not discuss how to implement the underlying encryption algorithm (AES) on programmable switches.

Chen [7] implemented the AES block cipher algorithm on the Barefoot Tofino programmable switch. We note that it is possible to truncate the output of AES and use it as a hash function with fixed input size. However, running AES is much slower than HalfSipHash. Specifically, the Tofino switch can only run 85 million AES-128 calculations per second for 16-byte inputs, compared with 150 million hashes per second with HalfSipHash-2-4 under the same setting.

To protect privacy, PINOT [27] encrypts and decrypts packet source and destination IP addresses on the Tofino switch. PINOT uses a simplified two-round Even-Mansour cipher that completes within a single pipeline pass and uses three independent keys. We note that a standard Even-Mansour cipher construction requires more rounds, which can similarly be implemented via packet recirculations. Even-Mansour cipher construction with two rounds using only one key is subject to key-recovery attacks [13].

7 CONCLUSION

We observe that unsafe hash functions are widely used in many network applications, leading to unintended security vulnerabilities. Thus, we present SipID: an optimized HalfSipHash-2-4 implementation for programmable switches in the data plane. Our evaluation on the Barefoot Tofino switch shows that SipID can calculate over 300 million hashes per second for 8-byte inputs. To the best of our knowledge, this is the first such implementation of its kind, and we envision SipID as the groundwork for secure hashing on many future network applications.

ACKNOWLEDGMENTS

This research is supported in part by DARPA contract HR0011-20-C-0160. We sincerely thank the anonymous reviewers as well as Mary Hogan, Hyojoon Kim, Robert MacDavid, and Jennifer Rexford for their thoughtful comments and feedback on this paper. We also thank Mengying Pan and Liang Wang for insightful discussions about CRC32's weaknesses and their valuable feedback.

REFERENCES

[1] Jean-Philippe Aumasson. 2021. SipHash Reference Implementation. https://github.com/veorq/SipHash. (2021). Accessed: 2021-5-13.

[2] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: A Fast Short-Input PRF. *Lecture Notes in Computer Science* 7668 (2012).

[3] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Santa Clara, CA, 667–683.

[4] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. 2020. Routing Oblivious Measurement Analytics. *IFIP Networking Conference (Networking)* (2020).

[5] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[6] 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. In *ACM SIGCOMM Conference*. 99–110.

[7] Xiaoqi Chen. 2020. Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables. *Workshop on Secure Programmable Network Infrastructure* (2020).

[8] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-grained queue measurement in the data plane. *International Conference on Emerging Networking Experiments And Technologies* (2019).

[9] Xiaoqi Chen, Hyojoon Kim, Javed M. Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP Round-Trip Time in the Data Plane. *Workshop on Secure Programmable Network Infrastructure* (2020).

[10] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. *ACM SIGCOMM* (2020).

[11] Frederik Hauser, Marco Haberle, Mark Schmidt, and Michael Menth. 2020. P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN. *IEEE Access* 8 (2020), 139567–139586.

[12] Frederik Hauser, Mark Schmidt, Marco Haberle, and Michael Menth. 2020. P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection With MACsec in P4-Based SDN. *IEEE Access* 8 (2020), 58845–58858.

[13] Takanori Isobe and Kyoji Shibutani. 2017. New key recovery attacks on minimal two-round Even-Mansour ciphers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 244–263.

[14] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache. *Symposium on Operating Systems Principles* (2017).

[15] Michael K. Kissi and Michael Asante. 2001. Penetration Testing of IEEE 802.11 Encryption Protocols using Kali Linux Hacking Tools. *International Journal of Computer Applications* 1761, 32 (2001), 26–33.

[16] Garam Lee, Hwajeong Seo, Taehwan Park, and Howon Kim. 2017. Optimized implementation of chaskey MAC on 16-bit MSP430. *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)* (2017).

[17] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium*. USENIX Association.

[18] Florian Mendel, Christian Rechberger, and Martin Schläffer. 2009. MD5 is weaker than weak: Attacks on concatenated combiners. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 144–161.

[19] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad. *ACM SIGCOMM* (2017).

[20] Inada Naoki. 2017. Moving to SipHash-1-3. https://bugs.python.org/issue29410. (2017).

[21] Tenkasi V Ramabadran and Sunil S Gaitonde. 1988. A tutorial on CRC computations. *IEEE micro* 8, 4 (1988), 62–75.

[22] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmuller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. 2019. Cryptographic Hashing in P4 Data Planes. *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2019).

[23] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. 2008. MD5 considered harmful today, creating a rogue CA certificate. In *25th Annual Chaos Communication Congress (CCC)*.

[24] Marc Stevens, Arjen Lenstra, and Benne De Weger. 2007. Chosen-prefix collisions for MD5 and colliding X. 509 certificates for different identities. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1–22.

[25] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. 2006. Birthday paradox for multi-collisions. In *International Conference on Information Security and Cryptology*. Springer, 29–40.

[26] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. *ACM SIGMOD International Conference on Management of Data* (2020).

[27] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. 2021. Programmable in-network obfuscation of DNS traffic. In *NDSS: DNS Privacy Workshop*.

[28] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 19–35.

[29] Sokolov Yura. 2015. Change Siphash to use one of the faster variants of the algorithm (Siphash13, Highwayhash). https://github.com/rust-lang/rust/issues/29754. (2015).

[30] Sokolov Yura. 2016. Switch SipHash from SipHash24 to SipHash13 Variant. https://github.com/ruby/ruby/pull/1501. (2016).

[31] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, Jianping Wu, and et al. 2020. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches. *Network and Distributed System Security Symposium* (2020).