Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

ABSTRACT

AES is a symmetric encryption algorithm widely used in many applications. An AES implementation in the data plane can help us build in-network security and privacy applications, such as IP header encryption or onion routing. However, it is not straightforward to implement AES on today's commodity programmable switches, which may not include a dedicated cryptography co-processor and support only simple arithmetic operation and table lookup. In this paper, we present the Scrambled Lookup Table technique for reducing the number of sequential arithmetic operations required for AES encryption, by utilizing the table matching capability available on programmable switches. We demonstrate an efficient implementation of AES on the Barefoot Tofino programmable switch that encrypts 10.92Gbit, 8.76Gbit, and 7.37Gbit of data per second, for AES-128, -192, and -256 respectively, using less than 15% of the switch memory.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; • **Security and privacy** → *Block and stream ciphers.*

KEYWORDS

Data Plane, P4, AES

ACM Reference Format:
Xiaoqi Chen. 2020. Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables. 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/10.1145/3405669.3405819

1 INTRODUCTION

Secure cryptographic primitives are critical building blocks for today's secure and private network protocols. Currently, these protocols are often executed in an end-to-end fashion, while network middle-boxes are often considered as a threat vector for man-in-the-middle attacks. As a consequence, cryptographic primitive functions like symmetric ciphers and hash functions are often run on end hosts, either on general-purpose CPUs or on dedicated cryptographic accelerator co-processors. More complex use cases beyond end-to-end security, such as the anonymity-preserving Onion Routing network (Tor), are often run in an overlay network.

The prevalence of high-throughput programmable switches has enabled researchers to propose various ways these devices can improve the privacy and security of our networks. For example, we can design efficient Tor-like anonymous onion routing directly between switches, protect user traffic from tampering [5] and surveillance [4], or make network measurements more secure [13] and anonymous [7]. Yet, early implementation of these ideas on programmable switches used custom-built cryptography functions that did not achieve the highest level of security, enabling powerful adversaries to attack the in-network application. For example, when a traffic anonymizer [7] uses CRC32 as its hash function, an adversary may exploit the linearity of CRC32 and launch a known-plaintext attack to breach data anonymity.

One reason researchers use unsafe cryptography primitives in their prototypes might be that implementing standardized cryptography primitives on programmable switches is *perceived* as requiring too much engineering effort and/or costing too much of the limited hardware resources available. In fact, as far as we know, there is no published implementation of any standardized secure cryptographic primitives for P4 programmable switches. Such implementation will help improve the security against adversaries for many existing security-oriented data-plane applications.

Advanced Encryption Standard (AES), previously known as the Rijndael algorithm [12] before its standardization, is one of the most widely used symmetric cryptography algorithms today, supporting countless secure protocols and network applications. It is a block cipher with 128-bit blocks, meaning the algorithm encrypts 128 bits of cleartext into 128 bits of ciphertext at a time; different variants of AES use different encryption key sizes, including 128 bits, 192 bits, and 256 bits. The algorithm consists of multiple encryption rounds (10, 12, or 14), each having four steps; we defer the details to Section 2. Due to its popularity, modern CPUs have special instruction sets (e.g., AES-NI [14]) specifically for computing one round of AES encryption. In the meantime, many efforts have been made to adapt AES into resource-constrained hardware environments, such as embedded systems with power constraints or memory constraints, but none are directly applicable to the execution environment of a programmable switch.

In this paper, we notice that a programmable switch imposes strict computational constraint, while having more memory than embedded systems. We therefore present the **Scrambled Lookup Table** technique for implementing AES, tailored for programmable network switches using the Reconfigurable Match-action Table (RMT) architecture [2]. We exploit the relatively abundant table

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPIN '20, August 14, 2020, Virtual Event, NY, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8041-6/20/08…$15.00
https://doi.org/10.1145/3405669.3405819

lookup memory on RMT switches and create many different permuted lookup tables using the encryption key, to further reduce the number of XOR operations required per encryption round.

With the Scrambled Lookup Table technique, we successfully implement the AES algorithm on a Barefoot Tofino Wedge-32X programmable switch. With the limited number of hardware pipeline stages available, our prototype implementation can calculate two AES rounds per pipeline pass for one 16-byte data block; it uses packet recirculations to let a data packet traverse the pipeline multiple times. Thus, we can complete AES-128 encryption (10 rounds) for a 16-byte payload in five pipeline passes, or AES-192/-256 (12/14 rounds) in six or seven pipeline passes, respectively. As our prototype implementation only uses a modest fraction of the switch memory, it can be combined with other networking functionalities, acting as a building block for designing more sophisticated secure and private network architectures built upon commodity programmable switches.

The rest of this paper is structured as follows. We discuss more details about AES algorithm, existing optimizations, and the hardware constraints of RMT-based programmable switches in §2. In §3, we present the Scrambled Lookup Table technique that optimizes our AES implementation for the switch data plane. §4 presents a working prototype of our AES implementation on a Barefoot Tofino programmable switch, and §5 shows experimental evaluation for the prototype. §6 discusses future works and §7 concludes the paper.

2 BACKGROUND

In this section, we introduce the AES algorithm and some existing hardware-friendly optimizations in more detail, as well as provide a brief overview of the programming model of the RMT programmable switch architecture.

2.1 The AES Algorithm

In 2001, the National Institute of Standards and Technology (NIST) selected Rijndael algorithm as the Advanced Encryption Standard (AES) [11]. AES was designated to be used by the US government to encrypt sensitive data, and later also used for classified data; it is also widely used by many commercial and open-source software.

AES [11] is a symmetric block cipher algorithm: given the encryption key, it encrypts a 128-bit (16-byte) data block into a 128-bit ciphertext; it can also decrypt the ciphertext block back to cleartext data, when the same encryption key is provided. AES supports different sizes of encryption keys (128, 192, or 256 bits), and these variants are commonly referred to as AES-128, AES-192, and AES-256.

The AES algorithm encrypts a data block by repeatedly applying encryption "rounds". The 16-byte data block is organized as a 4-by-4 matrix, with each cell holding one byte. Each encryption round consists of the following four steps, as illustrated in Figure 1:

(1) **`AddRoundKey`**: The data block is XORed with a key block, cell by cell. The round-specific key block is derived from the encryption key and the round number.
(2) **`SubBytes`**: Each byte in the block is replaced through a fixed lookup table, called the Substitution box (S-box).
(3) **`ShiftRows`**: Each row in the data block is cyclically shifted by 0, 1, 2, or 3 locations.
(4) **`MixColumns`**: The four bytes in each column are interpreted as a polynomial under the finite field $GF(2^8)$, and multiplied by a special polynomial $3x^3 + x^2 + x + 2$; the result (modulus $x^4 + 1$) is interpreted as the four bytes of the output column.

The algorithm completes after repeating the encryption round for 10, 12, or 14 times (for AES-128, -192, and -256 variants respectively), with minor changes in the last round. The three variants only differ in their key sizes and number of rounds required, while the procedure for encryption rounds remains the same. Decryption rounds are similar to encryption rounds, with the four steps performed in reverse and an inversed S-box. Due to space constraints, we defer the interested readers to the full algorithm proposal [11] for a more detailed description of the AES algorithm.

2.2 Prior AES Optimizations

We now briefly survey a few existing works on optimizing AES algorithms for other resource-constrained scenarios, including embedded devices and low-power circuits.

The authors of the Rijndael algorithm proposed the "T-table" approach for calculating AES rounds on 32-bit processors, which is discussed in Section 5.3.2 of the original Rijndael algorithm proposal [3]. Each AES round can be implemented using four lookup tables and XORs only, avoiding the expensive polynomial multiplications under $GF(2)$. This approach significantly accelerated the computation, at the cost of larger memory space required: it requires four lookup tables each using 256×4 bytes, while the original algorithm only requires 256 bytes for storing the S-box. Khairallah et al. [6] further optimized AES using lookup tables, making hardware-aware optimizations specific to the FPGA architecture. Luo et al. [9] also optimized AES using lookup tables for power-constrained use cases in wireless networks.

Embedded devices have very limited memory and computational power; in some cases, they cannot even store the entire S-box. Morioka and Satoh [10] and Wolkerstorfer et al. [15] present optimizations for efficiently calculating the S-box entries on the fly.

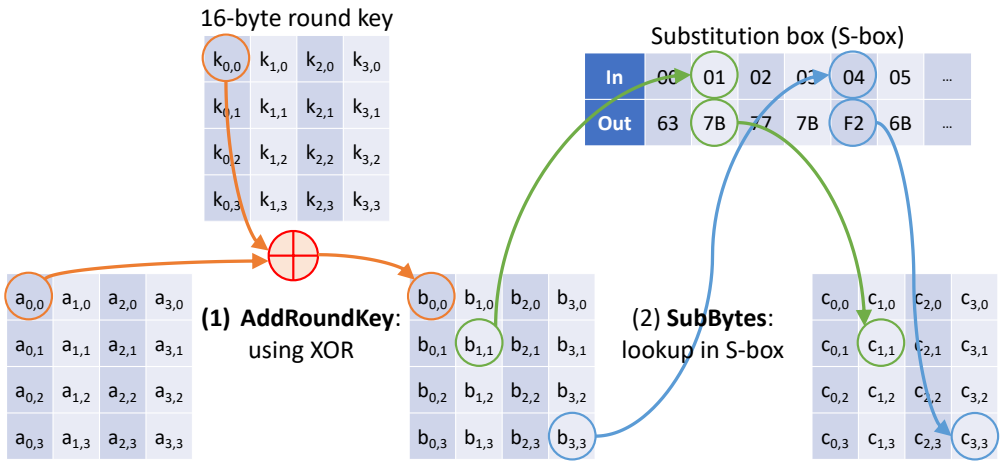
2.3 RMT Programmable Network Switches

Reconfigurable Match-action Table (RMT) is a programmable network switch architecture proposed by Bosshart et al. [2] that supports versatile yet high-performance network packet processing. In this section, we briefly describe the computation model of RMT switches.

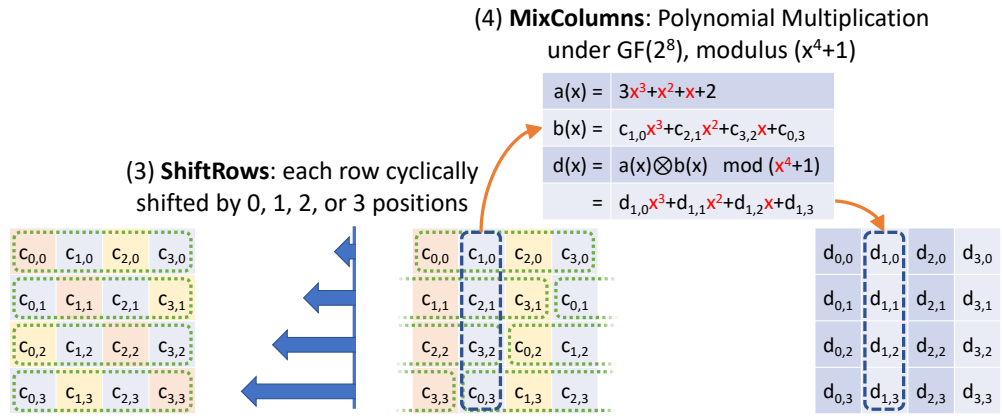
At a high level, a RMT switch first uses a programmable packet parser to parse program-specific protocol headers and extract header fields from the input packet, then uses a pipeline of match-action tables to match on those header fields and metadata variables and perform actions to modify them, and finally uses a deparser to emit the output packet.

A match-action table first *matches* on particular packet header fields or metadata variables, then takes a particular *action* to change some header fields or metadata variables. A match-action table hosts many match rules and their associated actions. In particular, an action may perform an arithmetic operation over metadata variables, including XORs.

The match-action pipeline consists of multiple pipeline stages, each of which supports executing multiple match-action tables in



(a) The AddRoundKey and SubBytes steps.



(b) The ShiftRows and MixColumns steps.

Figure 1: Illustration of one encryption round in the AES algorithm.

parallel. Multiple pipeline stages enable tables in later stages to match on the output of tables in earlier stages, so now we can compose more sophisticated functionalities. However, to achieve lower forwarding latency, practical programmable switches only have a limited number of hardware pipeline stages, typically 4 ∼ 32. For program that is overly complicated, it is possible to "recirculate" the packet to traverse the pipeline multiple times, in case one pipeline pass is not sufficient to finish all operations. The switch operator allocates a fixed fraction of bandwidth dedicated for recirculating packets; too many packet recirculations exhaust this bandwidth and lead to packet drops, thus we want to minimize the number of recirculations per packet.

We notice that the S-box lookup can be implemented using a match-action table, by matching on $b_{i,j}$s and writing different values to $c_{i,j}$s. Meanwhile, the XOR computations are supported on the programmable switches as well. Yet, naively translating the AES algorithm may lead to a complex data plane program that exhausts all pipeline stages, or requires too many recirculations. Therefore, we need to optimize our algorithm implementation for RMT architecture to minimize the number of pipeline stages needed.

3 OPTIMIZING AES FOR PROGRAMMABLE SWITCHES

Although the AES algorithm is modular and uses only simple arithmetic, a naive implementation may not fit into the limited hardware resources of a RMT programmable switch. In this section, we present the Scrambled Lookup Table technique for efficiently implementing AES encryption on RMT programmable switches.

Long Dependency Chain. We first notice that out of the four steps in the encryption rounds, both `AddRoundKey` and `MixColumns` can be implemented using XORs, and `ShiftRows` is essentially renaming variables. Furthermore, an existing AES implementation using the T-table construction can combine S-box and the effect of variable renaming and polynomial multiplications. Therefore, one AES encryption round can be simply implemented by four lookup tables and many XORs.

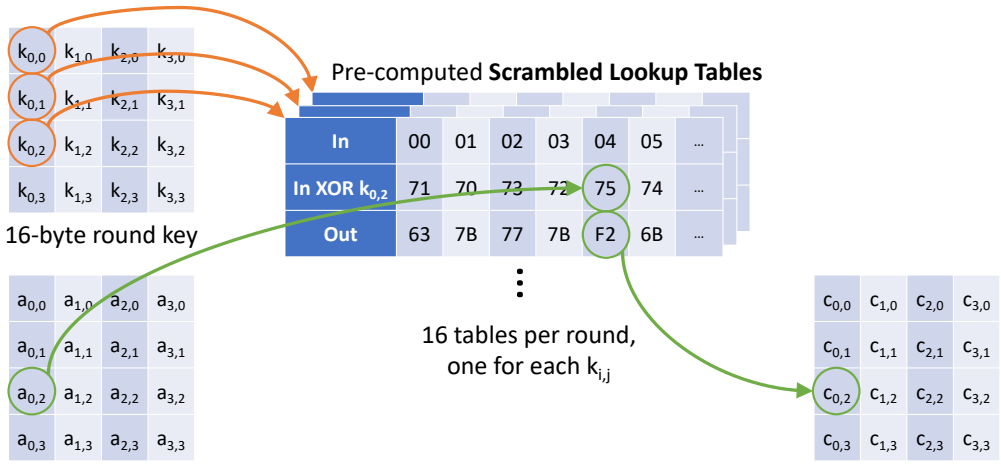


Figure 2: With Scrambled Lookup Tables, one lookup completes both the `AddRoundKey` and `SubBytes` steps.

Although RMT programmable switches can perform many arithmetic operations in parallel in each pipeline stage, the number of pipeline stages are limited. We want to shorten the number of arithmetic operations *in series*, or those creating dependencies one after the other. Unfortunately, in the traditional lookup table-based AES implementation, the `SubBytes` lookup depends on the XOR result of `AddRoundKey`, while we need to XOR many lookup results to obtain this round's output, before starting next round's `AddRoundKey`. This creates a rather long dependency chain (XOR-lookup-XOR), which translates to more hardware pipeline stages needed per encryption round.

Given that existing AES optimizations are geared towards embedded devices, which have a very small memory budget, we try a different approach. The S-box is only 256 bytes, and even in the optimized lookup table-based AES implementation storing lookup tables only costs 4 kilobytes. Programmable switches have abundant memory for match-action table entries, so we would like to inflate our implementation's memory footprint to trade for shorter data dependency, therefore using fewer hardware pipeline stages.

Use Multiple, Scrambled Lookup Tables. In Figure 2, we illustrate the idea of using Scrambled Lookup Tables to combine the `AddRoundKey` and `SubBytes` steps. Previously, one byte after the `SubBytes` step is calculated as:

$$c_{i,j} = Sbox[b_{i,j}] = Sbox[a_{i,j} \oplus k_{i,j}]$$

We now *scramble* the lookup table (Substitution box) for each (i, j), yielding this equivalent way of calculating $c_{i,j}$:

$$c_{i,j} = Scramble(Sbox, k_{i,j})[a_{i,j}].$$

To achieve this equivalence, we define the table scrambling operation as follows:

$$Scramble(Sbox, k)[a] = Sbox[a \oplus k], \forall a, k$$

We notice $f(a) = a \oplus k$ is a bijection. As the original Sbox lookup table has 256 entries, indexed from 0 to 255, we can prove that the scrambled lookup table also has the same 256 entries, for any scrambling key k between 0 and 255.

By permuting the lookup tables beforehand, we can complete both the `AddRoundKey` and `SubBytes` steps in one lookup, at the cost of storing 16 different lookup tables in each encryption round, as each byte now uses a different, scrambled lookup table.

After applying the Scramble Lookup Table technique, we follow the existing T-table optimization: the subsequent `ShiftRows` and `MixColumns` steps are folded into the lookup table step, by permuting the lookup table outputs and calculating multiplications beforehand. Now we simply need to XOR four different bytes together using three XORs to obtain $d_{i,j}$, which can immediately be fed into the Scramble Lookup Tables of the next encryption round. We also note that the Scramble Lookup Table technique can be similarly applied to AES decryption as well, however the encryption key is instead mixed into the output of lookup tables. In the next sections, we will show the extra memory required for the Scrambled Lookup Table technique is acceptable.

4 PROTOTYPE IMPLEMENTATION

In this section, we present our implementation of the AES algorithm running on a Barefoot Tofino Wedge32X programmable switch.

We implement a P4_16 program that takes UDP packets with 128-bit payload as input and emits the encrypted 128-bit ciphertext in the payload of output packets. In actual security or privacy applications, the encryption may be used over IP header fields (e.g., in ONTAS [7]). The parser of the P4 program merely parses Ethernet, IP, and UDP headers and subsequently parses the 128-bit payload into 16 individual bytes. Subsequently, the program performs two AES rounds in one pipeline pass, then recirculates the packet for more rounds until the encryption is finished. For each round, we use 16 Scrambled Lookup Tables to process each byte, then use 48 binary XOR operations to accumulate the lookup results. As the definition of these tables are very similar to each other, we used C-style macros to generate the source code for all these tables. The vanilla P4 source code is approximately 600 lines, and grows to approximately 950 lines after the macros are expanded. We have published the P4 source code and the corresponding control plane code on GitHub[1].

Besides the Scrambled Lookup Tables, the P4 program also uses a match-action table to decide if a packet should be recirculated,

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

based on the current encryption round. We add a temporary 4-byte header in the packet to maintain context, including the current encryption round and the final output port the packet should be routed to. By default, the Wedge32X switch has two 100Gbps loopback ports specifically reserved for packet recirculation. To maximize encryption throughput, we randomly choose one of the two ports for each packet selected for recirculation, by flipping a random fair coin using the switch's random number generator. If more ports are used for recirculation, we should change the random selection process accordingly to load-balance between these ports.

A python-based control plane is responsible for deriving the encryption key into round keys and installing matching rules for each scrambled lookup table. For example, when installing a new 128-bit encryption key for AES-128, the control plane derives 10 different round-specific key blocks; each round uses 16 scrambled lookup tables with 256 entries, so in total it installs $10 \times 16 \times 256$=40960 matching rules.

We verified our implementation's correctness using the test vectors available in the official AES specification [11].

5 EVALUATION

We now evaluate our prototype implementation of AES running on RMT programmable switches, by investigating its maximum throughput and hardware resource utilization.

5.1 Throughput

First, we measure the maximum encryption throughput we can achieve.

Setup: We connect two servers to one Barefoot Tofino Wedge32X 32-port programmable switch via 100Gbps Direct Attach Copper (DAC) cable. Each server has one Mellanox ConnectX-5 100G NIC and 20 CPU cores running at 2.2GHz. We use Pktgen-DPDK on one server to generate UDP packets with randomized source port and payload, and use a customized script on the other server to measure the packet receiving rate. Each UDP packet carries 128 bits (16 bytes) of payload, and has total size 58 bytes.

Recirculation bandwidth: Recall that the Wedge32X switch has two 100Gbps loopback ports reserved for recirculation, and the user can configure more physical ports to loopback mode to increase recirculation throughput. In our experiments, we leave the default configuration as-is, and measure the maximum encryption throughput we can achieve given the 2x100Gbps recirculation bandwidth.

We note that an incoming UDP packet is recirculated for a few times before leaving the switch to complete all encryption rounds, hence the switch will likely run out of recirculation bandwidth when the incoming packet rate is too high, leading to excessive packet drops. Therefore, we gradually increase the ingress packet rate and observe the peak egress packet rate, to find out the maximum possible encryption throughput under each setup.

Results: In Figure 3, we show the encryption throughput with respect to the incoming packet rate, for AES-128 algorithm (10 rounds). Here we present both an optimized version performing two rounds of encryption per pipeline pass (requiring 4 recirculations / 5 passes), as well as a baseline version performing only one round per pass (requiring 9 recirculations / 10 passes).

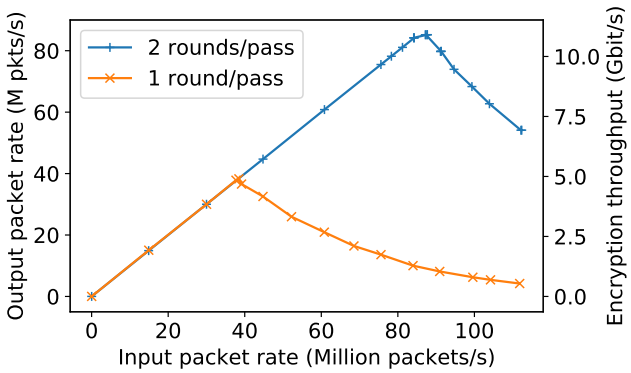


Figure 3: The prototype achieves maximum encryption throughput at 85 Million packets/second for AES-128, after which it starts to suffer from packet drops.

As we can see, the optimized prototype achieves the highest throughput at 85.3 Million packets per second (Mpps). Given that each 58-byte UDP packet carries a 128-bit encryption payload, this translates to 10.92Gbit/s (1.36GB/s) of encryption throughput. Meanwhile, the baseline version only achieves 4.91Gbit/s maximum throughput at 38.4 Mpps.

We repeat the experiment for AES-192 and AES-256 and observed similar patterns. In Figure 4, we plot the maximum throughput with respect to the number of total encryption rounds. As we expect, the throughput is inversely proportional to the number of recirculations required. Under the default 2x100Gbps loopback configuration, we can encrypt data at 8.76Gbit/s (1.10GB/s) for AES-192 and 7.37Gbit/s (922MB/s) for AES-256. To put these numbers into context, such throughput is on par with one modern desktop CPU core with instruction-set level AES optimization, or around four to six cores using software-based implementation.

We believe such throughput is sufficient for supporting a variety of in-network security and privacy innovations using programmable switches, such as IP header encryption and obfuscation; we anticipate these in-network applications only need to work on a fraction of bytes out of the entire line-rate traffic, as in the future most payload traffic will be protected by end-to-end encryption.

We further note that a higher encryption bandwidth is possible by reserving more switch ports as loopback; for example, if we reserve 50% of the ports, we can increase recirculation bandwidth by 8x and achieve a 8x increase in encryption throughput.

5.2 Hardware Resource Utilization

Now we analyze the resource footprint of AES encryption on programmable switches.

Memory: As we discussed earlier, the programmable switches have large memory for table matching, which is required for network routing. Technically, these exact match rules are stored in Static Random Access Memory (SRAM).

In our prototype implementation, each Scrambled Lookup Table stores 256 matching rules, and in total there are at most 16 tables times 14 rounds, i.e., 57k rules. Each rule costs 6 bytes to store (2 bytes matching + 4 bytes action data), which translates to 344 KB

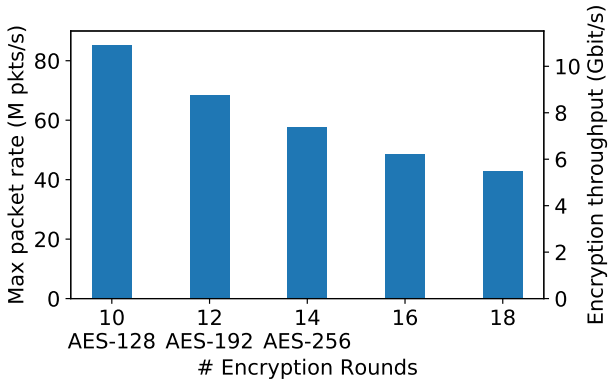


Figure 4: Maximum encryption throughput achieved under different number of encryption rounds.

for all rules. Even considering memory alignment and table over-provisioning, the actual memory footprint is still a modest fraction for today's switch hardware with tens of megabytes of SRAM.

Memory used for other switch functions (e.g., a couple of matching rules for deciding recirculation) is negligible. In total, we used less than 15% of the SRAM available on the hardware switch.

Tables: The programmable switch only supports a certain number of match-action tables. In our prototype, each encryption round is implemented using 16 Scrambled Lookup Tables, therefore a packet goes through 32 match-action tables to finish two rounds of encryption. The prototype used a couple more tables for various pre-processing and deciding recirculation. In total, it used less than 25% of the maximum number of match-action tables supported.

Arithmetic Operations: The programmable switch's packet processing pipeline supports a large, yet limited, number of instructions to perform in parallel in each pipeline stage, therefore we need to limit the number of arithmetic operations performed. In each encryption round in our prototype, the 16 Scrambled Lookup Tables output 16 four-byte values grouped into four rows. Within each row, we use three binary XOR operations to accumulate the 4 four-byte values into one. Therefore, we need 12 XORs in total to finish each encryption round, and need 24 XORs across the pipeline for two rounds of encryption.

Both the number of arithmetic operations used for encryption and for other miscellaneous processing are minimal, and the prototype used less than 10% of the maximum number of instructions supported.

In summary, our prototype implementation of AES algorithm did not extensively use any particular hardware resource on the programmable switch. It should coexist well with other switch functions when running together with other programs, or as a building block of a more sophisticated data plane program to implement secure and private network applications.

6 DISCUSSION

Encryption Mode: Our current prototype only encrypts one block of data at a time. For data longer than 16 bytes, naively encrypting each block individually is referred to as Electronic Code Book (ECB) mode, which is unsafe. Safer encryption modes are straightforward to implement, as they merely require an additional XOR for the data block with a counter, initialization vector, or an adjacent data block. These XORs are orthogonal to the block-level AES encryption.

Key Expansion: Currently, the AES key expansion procedure is completed in the switch control plane, and the key is installed in the form of tens of thousands of table entries for the Scrambled Lookup Tables. This process takes a few hundred milliseconds, and we could add versioning bits to the tables to maintain atomicity. It is also possible to compute the key expansion directly in the data plane, if more frequent key updates are desired.

Side-Channel Attacks: Encryption algorithms running on any hardware are subject to side channel attacks, such as key extraction through power or timing side channels. As the encryption key is represented by many table matching rules, an attacker may recover the key by probing the matching rules. We leave a more detailed hardware side channel analysis for the TCAM table matching mechanism for future work.

Performance: We note that our current AES implementation is a feasibility demonstration and not yet optimal for the Barefoot Tofino programmable switch, and it is possible to squeeze in three or four encryption rounds per pipeline pass. Also, we are only using the ingress half of the pipeline, while using the egress half can reduce the number of recirculation passes and potentially double the throughput.

However, we shall also bear in mind that there is only a limited room for further performance improvement. The match-action tables in RMT switches are not optimized for AES, thus can never achieve comparable throughput or power efficiency as dedicated CPU, FPGA, or cryptography co-processor. Alternatively, a programmable switch may add such dedicated cryptography circuit or co-processor, and allows P4 programs to use externs to compute cryptography functions, similar to what proposed in [5].

Aumasson [1] proposed using fewer encryption rounds in AES, to achieve higher throughput while providing practically acceptable security guarantee. Meanwhile, compared with AES, encryption algorithms using the Feistel cipher structure [8] can be implemented more efficiently using our Scrambled Lookup Table technique, as one encryption round (one lookup table and one XOR) can be finished within only one hardware pipeline stage. Thus, we also desire a data plane implementation of another standardized, widely-used encryption algorithm that uses Feistel cipher structure.

7 CONCLUSION

In this paper, we present the Scrambled Lookup Table technique for efficiently implementing AES encryption algorithm on RMT high-throughput programmable switches. We implement and evaluate a prorotype P4 program and show it can perform AES-128, AES-192, and AES-256 encryption at 10.92, 8.76, and 7.37 Gbit/s. This paves the way for implementing more sophisticated secure and private network protocols in the data plane of programmable switches.

8 ACKNOWLEDGMENTS

This research is supported by DARPA Contract No. HR001117C0047. We sincerely thank the anonymous reviewers of SPIN'20, as well as Jennifer Rexford, Tom Barbette, Hyojoon Kim, Praveen Tammana, Mary Hogan, Mengying Pan, and Robert MacDavid, for their helpful comments and feedback.

REFERENCES

[1] Jean-Philippe Aumasson. 2020. Too Much Crypto. *Real World Crypto Symposium (RWC 2020)* (2020).

[2] 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*.

[3] Joan Daemen and Vincent Rijmen. 1999. AES submission document on Rijndael, Version 2. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf. *National Institute of Standards and Technology, US Department of Commerce* (September 1999).

[4] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. 2019. SPINE: Surveillance Protection in the Network Elements. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*. USENIX Association, Santa Clara, CA.

[5] Frederik Hauser, Mark Schmidt, Marco Häberle, and Michael Menth. 2019. P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection with MACsec in P4-SDN. *arXiv preprint arXiv:1904.07088* (2019).

[6] Mustafa Khairallah, Anupam Chattopadhyay, and Thomas Peyrin. 2017. Looting the LUTs: FPGA optimization of AES and AES-like ciphers for authenticated encryption. In *International Conference on Cryptology in India*. Springer, 282–301.

[7] Hyojoon Kim and Arpit Gupta. 2019. ONTAS: Flexible and Scalable Online Network Traffic Anonymization System. In *2019 Workshop on Network Meets AI & ML*. 15–21.

[8] Lars R Knudsen. 1993. Practically secure Feistel ciphers. In *International Workshop on Fast Software Encryption*. Springer, 211–221.

[9] Xinqiang Luo, Yue Qi, Yadong Wan, Qin Wang, and Hong Zhang. 2014. A fast AES encryption method based on single LUT for industrial wireless network. In *International Conference on Identification, Information and Knowledge in the Internet of Things*. IEEE, 158–161.

[10] Sumio Morioka and Akashi Satoh. 2002. An optimized S-Box circuit architecture for low power AES design. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 172–186.

[11] National Institute of Standards and Technology. 2009. FIPS 197, Advanced Encryption Standard (AES). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf. *National Institute of Standards and Technology, US Department of Commerce* (November 2009).

[12] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. 2001. Report on the development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology* 106, 3 (2001), 511.

[13] Pedro Reviriego and Daniel Ting. 2020. Security of HyperLogLog (HLL) Cardinality Estimation: Vulnerabilities and Protection. *IEEE Communications Letters* (2020).

[14] Jeffrey Rott. 2010. Intel Advanced Encryption Standard instructions (AES-NI). https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni. *Technical Report, Technical Report, Intel* (2010).

[15] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. 2002. An ASIC implementation of the AES SBoxes. In *Cryptographers' Track at the RSA Conference*. Springer, 67–78.