An FPGA Implementation of SipHash

Benjamin Welte
Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
bwelte99@iastate.edu

Joseph Zambreno
Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
zambreno@iastate.edu

***Abstract*—Cryptographic hash functions play a critical role in ensuring the security and veracity of network transactions; for example, they constitute the backbone of hash-based message authentication codes (HMACs), distributed hash tables (DHTs), and blockchain. However, cryptographic hashing can incur significant CPU overhead, especially for applications that commonly process large inputs exceeding 1 MB. This can make it infeasible to implement HMACs, DHTs, etc. in resource-constrained embedded systems or servers with strict response time requirements. As a solution, we present an FPGA architecture to accelerate SipHash, a promising cryptographic hash function. Our design constitutes the first SipHash implementation that targets maximum performance on an FPGA. The proposed architecture's throughput and acceleration vs. software were measured on Xilinx's Zynq-7000 and Ultrascale+ SoCs for a wide range of input sizes. These results show one core can provide single-threaded throughput of up to 13.7 Gbps on a modern FPGA fabric, and multiple parallel cores can exceed 100 Gbps, allowing applications like blockchain and peer-to-peer file sharing to scale with emerging high-bandwidth networks. A single core can keep pace with 10 Gigabit Ethernet, and further parallelization can empower FPGA designs to fully utilize higher network bandwidths.**

I. Introduction

Hash functions are extremely versatile algorithms with many diverse uses. They compress (or "hash") inputs of varying length into outputs of fixed size. A subcategory of hash functions are cryptographic, meaning they take a security key as an additional input to encrypt their hash. In applications such as hash-based message authentication codes (HMACs) [1]–[3], distributed hash tables (DHTs) [4]–[6], and blockchain [7], [8], keyed hash functions offer security in addition to data compression and fault-tolerance. Their ubiquity has led noted cryptographer Bruce Schneier to comment, "Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography" [9]. These cryptographic hash algorithms have two desirable characteristics: speed and security.

However, achieving high speed in tandem with cryptographic strength is not trivial. Many secure hash functions struggle to keep pace with throughput demands, and this problem becomes exacerbated in the presence of large inputs. Applications like peer-to-peer file-sharing networks and blockchain regularly process inputs of megabytes or more, compelling a difficult choice between security and efficiency. This explains why insecure hash functions like MD5 and SHA-1 remain commonplace despite their well-documented vulnerabilities [10], [11]: designers have no choice but to use a fast, insecure algorithm when strict throughput demands make a more secure alternative's latency intolerable. Other more robust hash functions such as SHA-2 enable foundational services such as HTTPS and SSH [12], but the computational complexity that gives them cryptographic strength also limits their throughput. Many applications could benefit from a hash function implementation that combines high speed with cryptographic integrity. To achieve this goal, it is much more straightforward to accelerate a secure algorithm as opposed to modifying an insecure one.

In the past, exponential improvements in integrated circuit fabrication might have sufficiently accelerated hash functions like SHA-2. But Moore's Law, at least according to its original formulation, is dead. We can no longer rely on decreasing feature sizes to accelerate general purpose computing. As such, Horowitz and others have advocated specialized hardware as the best available means to continue improving performance and energy efficiency [13]. With respect to hash functions in particular, field-programmable gate arrays (FPGAs) offer a promising acceleration option. As Herbordt et al. point out, FPGAs excel at speeding up scalable, low-precision computations, and they have the flexibility to implement a family of algorithms as opposed to a single point solution [14]. The literature also contains many examples of leveraging FPGAs' programmable logic to explore tradeoffs between area and performance such as [15]. Modern FPGA fabrics are also tightly integrated with general purpose processors and various other hardware modules, promoting a healthy symbiosis between hardware and software. Consequently, various FPGA accelerators already implement common hash functions like MD5 [16] and SHA-2 [17]–[21] with varying emphases on speed, area, energy efficiency, etc. Such designs can achieve high and efficient throughput, but they leave algorithmic deficiencies unresolved. For instance, accelerating MD5 does not fix its inherent insecurity, and SHA-2's complexity incurs substantial power and utilization overhead on an FPGA.

To avoid these common algorithms' shortcomings and meet the demand for fast, secure hashing, we propose implementing a family of hash functions known as SipHash on an FPGA [22]. Unlike MD5, SipHash is cryptographically robust, and unlike SHA-2, its simple sponge construction relies wholly on ARX (add-rotate-xor) operations amenable to hardware acceleration. SipHash was originally intended as an HMAC

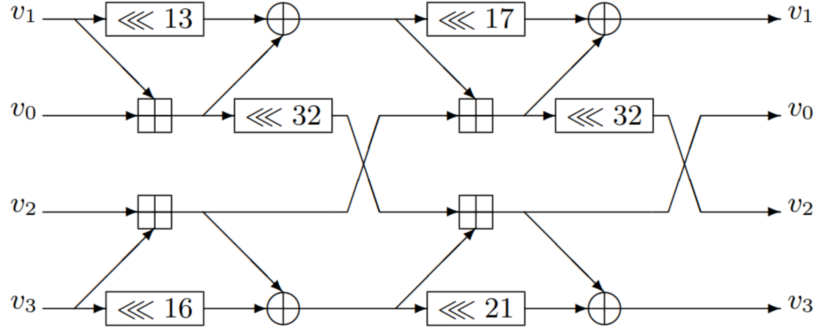

Fig. 1. The ARX Network of SipRound

and for creating flooding-resistant hash tables [22], so our architecture can service these applications as well as many others requiring fast and secure hashing.

II. SipHash

SipHash was first proposed by J.P. Aumasson and Daniel Bernstein in 2012 to replace MD5 as an HMAC and to combat hash-flooding denial of service attacks [22]. It consists of three stages:

1) **Initialization**: four internal state variables, v_0, v_1, v_2, and v_3, are initialized to algorithmic constants:

$$v_0 = k_0 \oplus \text{x736F6D6570736575}$$

$$v_1 = k_1 \oplus \text{x646F72616E646F6D}$$

$$v_2 = k_0 \oplus \text{x6C7967656E657261}$$

$$v_3 = k_1 \oplus \text{x7465646279746573}$$

Here, k_1 and k_0 are the upper and lower 64 bits of the security key, respectively, in little-endian format.

2) **Compression**: SipHash-c-d processes a b-byte string as $\lceil (b+1)/8 \rceil$ 64-bit little endian words, the last of which includes the final b % 8 bytes of the message followed by null bytes and terminated with a byte encoding of b % 256. For example, a 9-byte message consisting of x00, x01, x02, x03, x04, x05, x06, x07, and x08 would form two words: x0706050403020100 and x0900000000000008. To compress the message, the internal state iteratively absorbs each word, m_i, with these steps:

- $v_3 \oplus m_i$
- c iterations of SipRound
- $v_0 \oplus m_i$

The SipRound function, also pictured in Figure 1 from [22], consists of the following operations:

$$\begin{array}{cc} v_0 \mathrel{+}= v_1 & v_2 \lll 32 \\ v_1 \lll 13 & v_2 \mathrel{+}= v_3 \\ v_1 \oplus= v_0, & v_3 \lll 16 \\ v_0 \lll 32 & v_3 \oplus= v_2 \\ v_2 \mathrel{+}= v_1 & v_0 \mathrel{+}= v_3, \\ v_1 \lll 17, & v_3 \lll 21 \\ v_1 \oplus= v_2 & v_3 \oplus= v_0 \end{array}$$

Here, '$+$' denotes bitwise logical or, '$\oplus$' denotes bitwise exclusive or, and '$\lll$' denotes a left barrel shift. Obviously, these concurrent operations present an opportunity for parallelism in hardware.

3) **Finalization**: After compressing each message word, SipHash xors the constant xFF to the internal state, performs SipRound d more times, and then returns $v_1 \oplus v_2 \oplus v_3 \oplus v_4$ as the hash.

Figure 2 shows an illustration from [22] of SipHash2-4 hashing a 15-byte message. Bernstein and Aumasson recommend using SipHash2-4 for maximum security, but SipHash1-3 is also widely used. For more information on the SipHash algorithm, please see [22].

Compared to more popular algorithms like SHA-2, cryptanalysis of SipHash is in its infancy, but several preliminary studies present promising results. [23] did not uncover any viable differential characteristics for a key-extraction attack. The authors from [24] refine the methodology from [23] but similarly deem both SipHash2-4 and SipHash1-3 secure against differential cryptanalysis. Although successful attacks have been devised against reduced-round SipHash [25], these do not affect versions of the algorithm used in practice.

In terms of side-channel attacks, the authors of [26] demonstrate that an attacker can determine SipHash's security keys by monitoring the power consumption of a CPU executing it, although they note this is significantly more difficult and requires many more power probes than for other common algorithms like AES. Since this attack targets SipHash software, moving to an FPGA-based design would actually mitigate it. However, it might be possible to adapt an attack similar to the one outlined against SHA-3 in [27] to determine

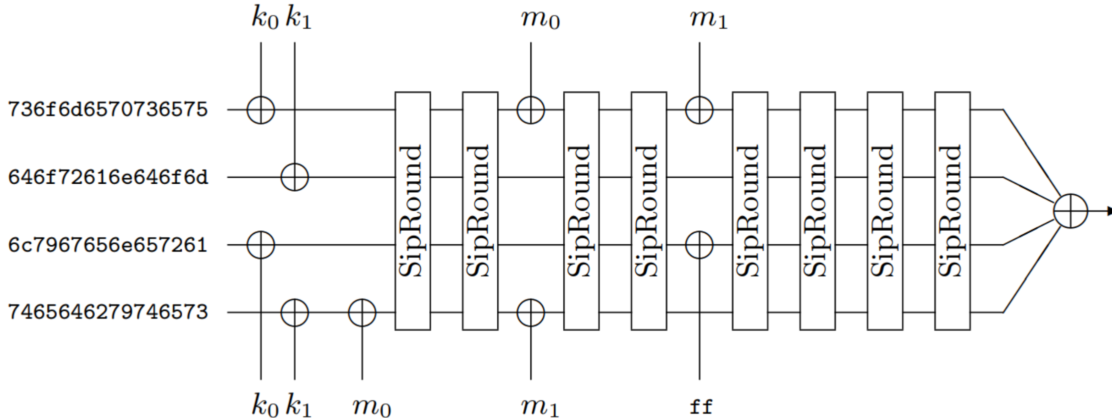

Fig. 2. SipHash2-4 Hashing a 15-byte Message

SipHash keys based on an FPGA's power consumption. But as the authors of [27] observe, such energy-based side-channel attacks become more difficult on boards with lower supply voltages. It therefore seems the electrical characteristics of the specific board a design is deployed on will determine its resiliency against power-based attacks more than the choice of a hashing algorithm.

III. Related Work

[28] details attempts by researchers at Google to accelerate SipHash using the AVX2 SIMD extensions to the x86 ISA, but this proved relatively unsuccessful due to AVX2's lack of vector shift instructions. The authors of [29] include an FPGA implementation of SipHash in their work outlining attacks and defenses for multi-tenant FPGA systems. However, their design's 32-bit interface can only load half of SipHash's 64-bit inputs during one clock cycle. As such, it can only approach throughput of 0.5 cycles/byte for long messages. [29] also loads the security key serially into the core over 4 clock cycles, indicating that it may have been intended as more of an ASIC prototype than an FPGA-specific architecture.

The literature contains few SipHash accelerators outside the works outlined above, but it does present numerous FPGA implementations of other hash functions, most notably SHA-2. These architectures differ in emphasizing energy efficiency [21], throughput per unit area [18], and a combination of both [20], but the algorithm's inherent complexity incurs painful tradeoffs between these metrics. The design in [21], for example, achieves reasonable throughput - about 2.5 Gbps - but at the cost of over 200,000 LUTs and 350,000 flip-flops. Furthermore, because of SHA-2's complexity, many of these designs accelerate SHA-2 with a fully programmable processor that only includes the functional units required for hashing [19]. By contrast, streamlined algorithms like SipHash can hash inputs without fetching and decoding instructions. In applications that don't specifically require SHA-2, it therefore seems prudent to choose a simpler hash function.

IV. Architecture

We present here the first SipHash design tailored to an FPGA. Figure 3 shows the proposed architecture which we implemented at the RTL level with VHDL. The notation for k_0, k_1, and m_i is kept from [22], and the variable names c_0, c_1, c_2, and c_3 denote the algorithmic constants that initialize the internal state. Four 64-bit registers store the intermediate values of v_0, v_1, v_2, and v_3 following each of the algorithm's initialization and compression phases. The internal state's initial and current values are multiplexed onto the datapath to appropriately update the state depending on whether hashing is just starting or already underway. To accomplish finalization, the internal state passes through a pipeline of d SipRounds before the state variables are xored together to produce the hash.

A hardware wrapper interfaces the core with two separate AXI-Stream buses, one which receives input data and the other which streams out the resulting hashes [30]. As such, control signals from the incoming slave AXIS interface (e.g. TLAST, TREADY, TVALID, etc.) demarcate valid input data, control the compression datapath's multiplexers, and ensure the register holding the final output will only latch valid hashes.

It is worth noting that this design assumes the final word in the input message has already been padded with zeros and the message size in bytes size modulo 256. This allows the core to dynamically handle inputs of various sizes determined by the position of TLAST on the input stream as opposed to requiring a configuration register or a similar recalibration mechanism to handle inputs of varying size. However, this does necessitate either software or an upstream unit to pad the last word before it reaches the core. Since hash functions typically have a large ratio of input to output data, this design assumes each hash will be read via either hardware or software before the next valid hash overwrites it.

This design also includes several registers that allow a modern FPGA's processing subsystem to easily configure it. In

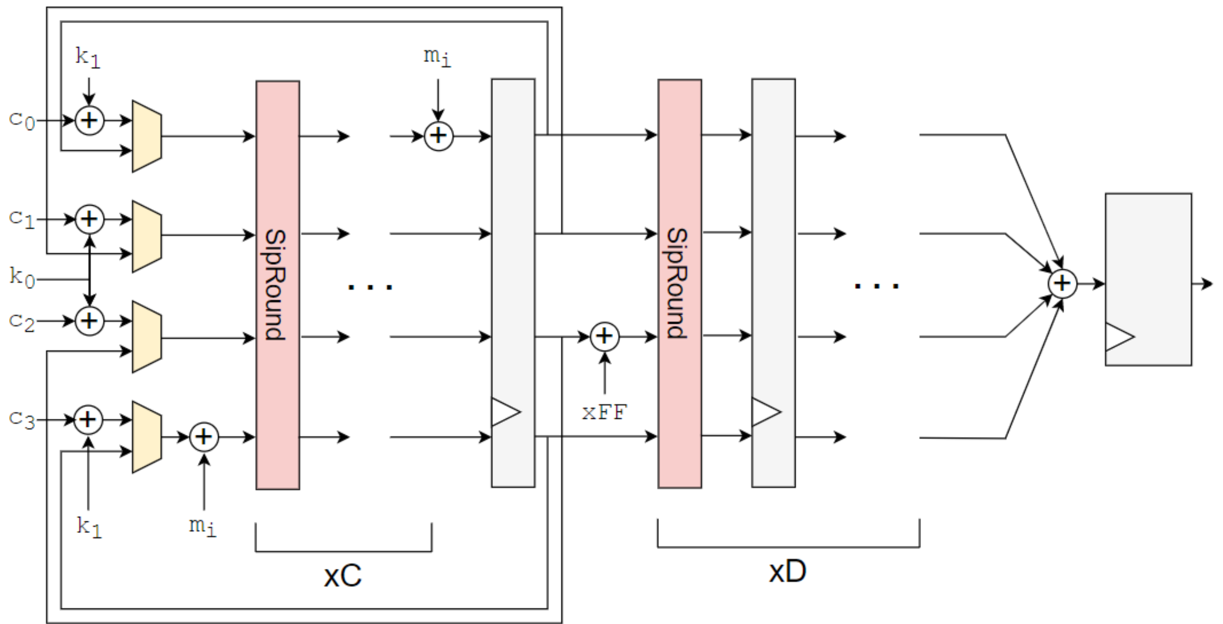

Fig. 3. Proposed SipHashC-D Architecture

Register No.	Offset	Description
0	x0	k0 [31:0]
1	x4	k0 [63:32]
2	x8	k1 [31:0]
3	x12	k1 [63:32]
4	x16	Soft Reset (Active high; only 0th bit)
5	x20	Hash Count
6	x24	Hash Value [31:0]
7	x28	Hash Value [63:32]

TABLE I
SIPHASH CORE CONFIGURATION REGISTERS

our implementation, an embedded processor can read and write to these registers using the AXI-Lite protocol [31]. Table IV describes the memory-mapped registers in detail. They allow software to reset the core, initialize the SipHash keys, and read the last valid hash as well as the number of valid hashes since the last reset. The core's output is therefore accessible at both the RTL and application levels.

Theoretically, this architecture will achieve throughput of $\frac{w+d}{8w}$ cycles per byte where w is the number of 64-bit words in the message and d is the number of finalization rounds. This follows from straightforward analysis: if the design processes one 64-bit word (8 bytes) each cycle, it will produce a valid hash d cycles after compressing the last word. If w is much larger than d or if the core receives multiple back-to-back inputs, this will hide the extra latency introduced by the pipeline, allowing throughput to asymptotically approach 0.125 cycles/byte. The design's throughput therefore depends primarily on the fastest possible clock frequency in its critical path, the recursive loop connecting the internal state register's output to its input. Further pipelining of this region is impossible because the output of each compression must become available during the same cycle as the next input, so each pipeline stage during compression would require delaying each input word by a cycle, defeating the purpose of a pipeline. Unrolling this loop to further pipeline it as in [15] would irrevocably tether the design to a fixed input size. As such, leaving the critical path in the compression stage sacrifices some performance while allowing the architecture to hash inputs of arbitrary length.

V. EVALUATION METHODOLOGY

We evaluated the proposed architecture based on its throughput and resource utilization on an FPGA. We are particularly interested in our architecture's performance on large inputs, e.g. 2^{12} bytes and greater. Large input sizes naturally cause hashing to consume more CPU time and make its acceleration more beneficial in accordance with Amdahl's Law. Applications like IPSec that use HMACs may need to hash messages up to 2^{13} bytes in size [2], and inputs to blockchain and peer-to-peer file sharing systems can easily exceed 1 MB [5], [8].

We profiled our design using Xilinx's Zynq-7000 and Ultrascale+ SoCs to gauge its performance on edge and server devices. The specific boards we chose that contained each FPGA were the Zedboard and the ZCU-106, respectively. This provided comparisons between our SipHash core implemented on multiple generations of programmable logic fabric, the Zynq-7000's ARM Cortex-A9 CPU, and the Ultrascale+'s more performant ARM Cortex-A53.

We measured resource utilization after synthesizing this design for the aforementioned platforms. Table II contains these results as well as the design's fastest possible clock frequency on both boards. Throughput was profiled using a

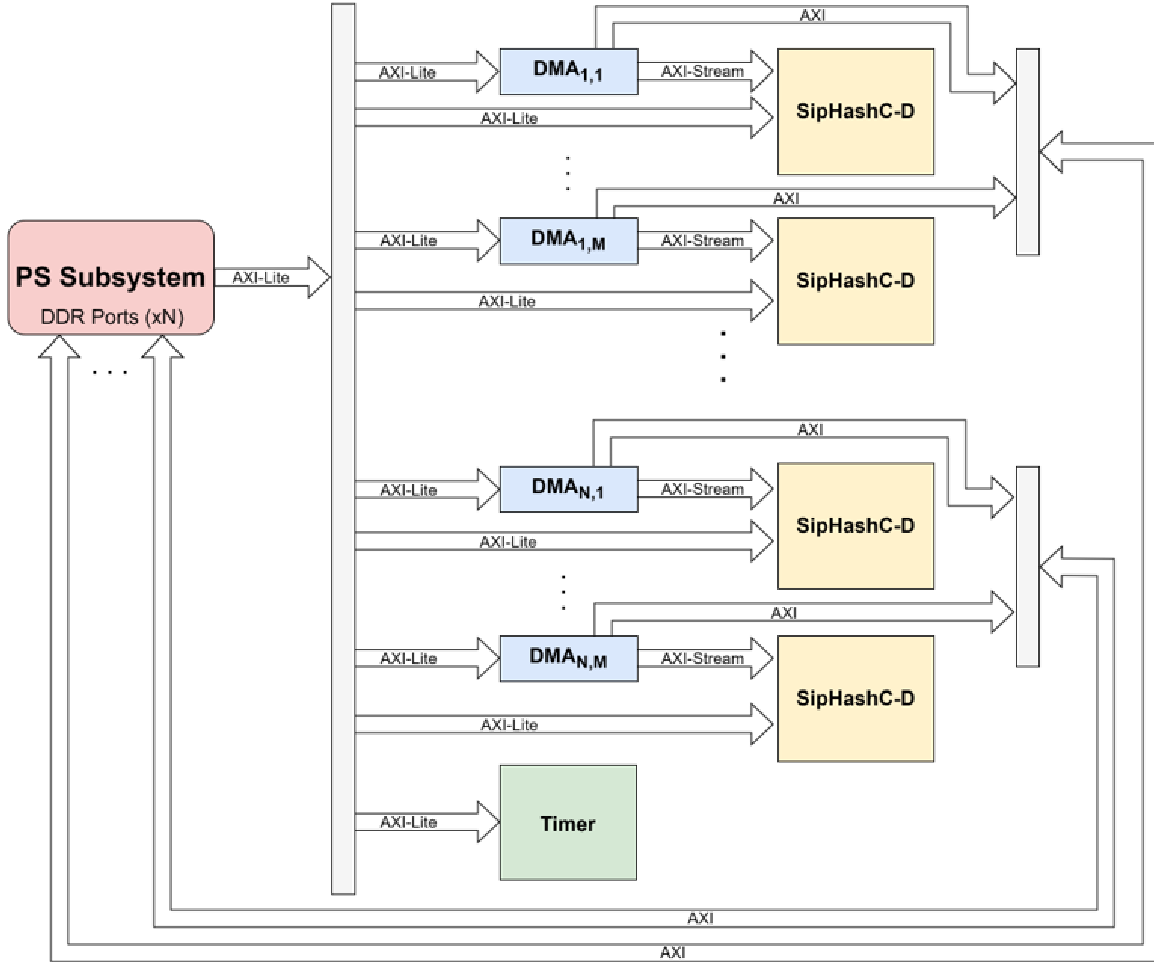

Fig. 4. Hardware Test Bench

TABLE II
UTILIZATION AND MAX CLOCK FREQUENCIES

Board	Algorithm	Max Clk. Frequency	LUTs (% total)	Registers (% total)
Zedboard	SipHash2-4	71.4 MHz	2397 (4.5%)	1396 (1.3%)
Zedboard	SipHash1-3	111.1 MHz	1598 (3.0%)	1138 (1.1%)
ZCU-106	SipHash2-4	214.3 MHz	2355 (1.02%)	1395 (0.3%)
ZCU-106	SipHash1-3	214.3 MHz	1594 (0.69%)	1141 (0.25%)

test bench comprised of complementary hardware and software available on Github [32]. Figure 4 shows the hardware test bench; it instantiates multiple SipHash cores, each with an accompanying DMA to stream input vectors from off-chip DRAM. Software initiates DMA transactions to stream test vectors to each SipHash core before polling its result register to check if hashing has completed. This application also times the core by reading a hardware timer instantiated in the programmable logic fabric to measure the clock cycles that elapse while the core generates each hash. The processing subsystem can then similarly measure the number of PL timer cycles it needs to execute Bernstein and Aumasson's open source software implementation of SipHash [33] compiled with -O3 optimization. This scheme fairly compares the time required to initiate and complete hashing in both hardware and software. We repeated this experiment on both boards with an increasing number of cores until adding additional cores no longer increased the total throughput.

VI. RESULTS

The tests described in Section V were run using input vectors ranging in size from 8 bytes to 1 MiB. Figure 5 shows the design's throughput and acceleration vs. software on both the Zedboard and the ZCU-106. (Since SipHash1-3 only performs as well or worse than SipHash2-4 on the ZCU-106, we omit a graph for those results). These graphs approach asymptotes as the input sizes increase because hardware hashing includes the latency required to initiate each DMA transaction and read the PL timer. For small inputs, these

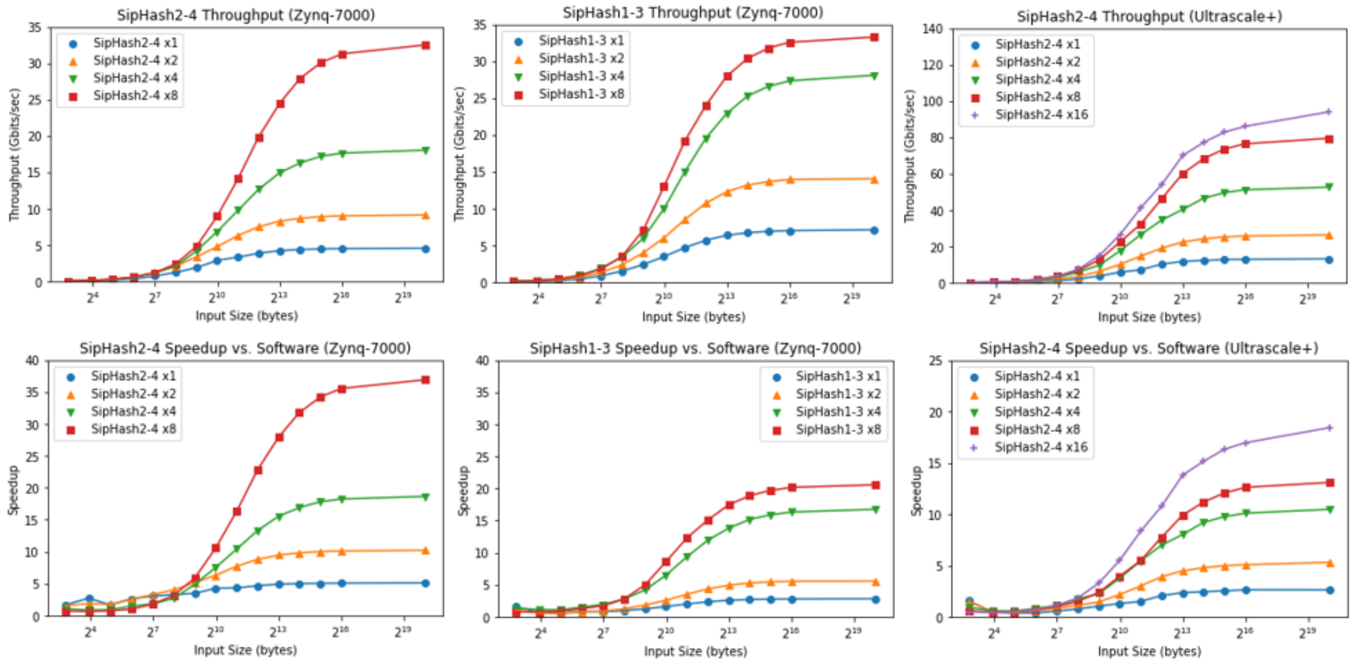

Fig. 5. Throughput and Acceleration vs. Software

TABLE III
SIPHASH THROUGHPUT AND ACCELERATION

Board	Algorithm	SipHash Cores	Throughput (Gb/sec)	Speedup vs. Software
Zedboard	SipHash2-4	1	4.57	5.12
Zedboard	SipHash2-4	2	9.14	10.23
Zedboard	SipHash2-4	4	18.03	31.41
Zedboard	Siphash2-4	8	32.49	36.89
Zedboard	SipHash1-3	1	7.11	2.83
Zedboard	SipHash1-3	2	14.04	5.58
Zedboard	SipHash1-3	4	28.08	17.36
Zedboard	SipHash1-3	8	33.25	20.56
ZCU-106	SipHash2-4	1	13.18	2.62
ZCU-106	SipHash2-4	2	26.36	5.31
ZCU-106	SipHash2-4	4	52.59	10.46
ZCU-106	SipHash2-4	8	79.40	13.08
ZCU-106	SipHash2-4	16	93.86	18.41
ZCU-106	SipHash1-3	1	13.18	1.60
ZCU-106	SipHash1-3	2	26.36	3.14
ZCU-106	SipHash1-3	4	52.59	6.23
ZCU-106	SipHash1-3	8	76.30	7.14
ZCU-106	SipHash1-3	16	92.45	10.89

TABLE IV
COMPARISON TO OTHER WORK

Algorithm	Throughput (cycles/byte)	Max Clk. Frequency (MHz)	LUTs	Registers
SipHash2-4 [28]	1.26	N/A	N/A	N/A
SipHash1-3 [28]	0.68	N/A	N/A	N/A
SipHash2-4 [29]	0.5	173.0	907	789
SipHash2-4 [us]	**0.125**	**214.3**	**2355**	**1395**
SipHash1-3 [us]	**0.125**	**214.3**	**1594**	**1141**

overheads dwarf the hash time itself, making throughput and acceleration poor. However, as the inputs become larger, these overheads become less significant, improving performance. We are primarily interested in accelerating hashing for large inputs ($\geq 2^{12}$ bytes), and the overhead from initializing the DMA is insignificant in these cases. We therefore report the core's asymptotic throughput and acceleration for both boards in Table III.

Table II shows the resource utilization and maximum clock frequency for SipHash2-4 and SipHash1-3 on both the Zynq-7000 and Ultrascale+. SipHash1-3 requires fewer resources than SipHash2-4 because it instantiates fewer SipRounds in the core's compression and finalization stages. On the Zynq-7000, this also reduces the critical path by downsizing the unpipelineable compression loop, increasing the maximum clock frequency. However, on the Ultrascale+, the test bench's critical path resides in the DMAs. In practice, this causes SipHash2-4 and SipHash1-3 to have the same max clock frequency and therefore the same throughput on the ZCU-106.

A single core implementing either SipHash2-4 or SipHash1-3 on the Zynq-7000 FPGA can easily hash packets received from Gigabit Ethernet faster than they come in, and on the Ultrascale+, one core can outpace 10 Gigabit Ethernet by itself. Deploying multiple parallel cores further improves performance; Figure 6 depicts total and average throughput as a function of the number of parallel cores in use. Both FPGAs only offer four DRAM ports, so multiple cores must share a port if $N \geq 4$. This explains the dropoff in average throughput we usually observe each time the number of cores is incremented past a multiple of four: such increases also increment the largest number of cores sharing an input bus.

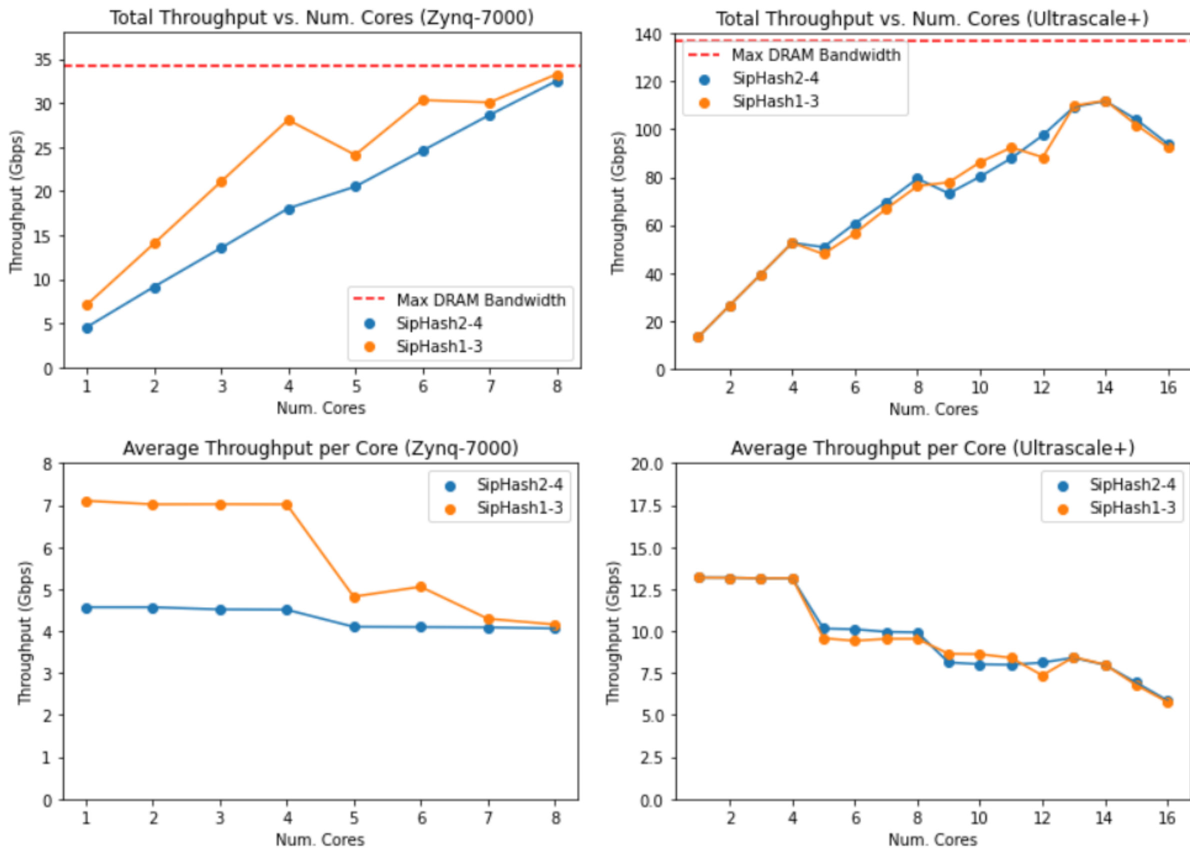

Fig. 6. Total and Average Throughput per Core

On the Zedboard, eight cores implementing either SipHash2-4 or SipHash1-3 achieve throughput of up to 34 Gb/sec on the Zynq-7000, maximizing the bandwidth of the Zedboard's DDR3. Since SipHash1-3 is faster on the Zedboard, five cores are actually sufficient to reach the memory bandwidth bound and cause a steep decrease in average throughput for five cores compared to four. On the Ultrascale+, performance is limited by contention for the finite number of DRAM ports instead of the DDR's bandwidth, but peak performance still exceeds 100 Gb/sec, reaching 111 Gb/sec with fourteen parallel cores before more units create bus contention that decreases the total throughput.

Table IV compares our design to other examples of SipHash acceleration. For SipHash2-4, the proposed architecture achieves nearly ten times the throughput per cycle of the AVX2 implementation in [28] (this is the only throughput measurement they provide). It also achieves four times the throughput per cycle of the FPGA implementation in [29] with about 2.6 times as many LUTs and 1.8 times as many registers. It achieves similar numbers for SipHash1-3, but its acceleration vs. the AVX2 implementation is less pronounced. This is consistent with Table III which shows that the core accelerates SipHash1-3 less than SipHash2-4 compared to software.

To gauge our design's viability in a server environment, we also compared our results to the performance of a 12th generation Intel i7 Alder Lake CPU [34] executing both SipHash2-4 and SipHash1-3. The Alder Lake CPU's throughput varied widely across runs, but on average, it achieved roughly 11.08 Gbps throughput for SipHash2-4 and 13.04 Gbps for SipHash1-3. These numbers are only slightly lower than our FPGA design's 13.70 Gbps mark for both algorithms on the ZCU-106. However, it's worth noting that the Ultrascale+ uses 8th generation (28 nm node) technology whereas the Alder Lake uses 12th gen (14 nm). Clearly, this skews the comparison in favor of the Alder Lake. Since we did not have an 8th generation CPU to compare against, future work could better contextualize our design's performance compared to a server CPU from a similar generation.

VII. Conclusion

In this paper, we present a novel design capable of implementing any algorithm from the SipHash family on an FPGA. This architecture will allow embedded FPGA-based systems to securely and performantly hash incoming and outgoing network traffic in a wide range of applications. Experimental results show the design can outpace Gigabit Ethernet using just one core, and deploying multiple cores in parallel can generate throughput exceeding 100 Gigabits per second. This represents a marked improvement over related accelerators in

the literature for both SipHash and other hash functions like SHA-2. Future work might investigate architectural improvements such as implementing SipHash with fewer resources as well as comparing this design to a conventional server using an FPGA and a CPU fabricated with the same technology process.

References

[1] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," 1997.

[2] S. I. Naqvi and A. Akram, "Pseudo-random key generation for secure HMAC-MD5," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, 2011.

[3] O. Elkeelany, M. Matalgah, K. Sheikh, M. Thaker, G. Chaudhry, D. Medhi, and J. Qaddour, "Performance Analysis of IPSec Protocol: Encryption and Authentication," in *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No.02CH37333)*, 2002.

[4] H. Zhang, Y. Wen, H. Xie, and N. Yu, *Distributed hash table: Theory, Platforms and Applications*. Springer, 2013.

[5] A. Rosen, B. Levin, and A. G. Bourgeois, "Autonomous load balancing in distributed hash tables using Churn and the Sybil attack," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2001.

[6] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big learning NIPS workshop*, 2013.

[7] A. H. Lone and R. Naaz, "Demystifying cryptography behind blockchains and a vision for post-quantum blockchains," in *2020 IEEE International Conference for Innovation in Technology (INOCON)*, 2020.

[8] S. Jiang and J. Wu, "Bitcoin mining with transaction fees: A game on the block size," in *2019 IEEE International Conference on Blockchain (Blockchain)*, 2019.

[9] B. Schneier, "Encryption must move beyond secure hash algorithm," *Network World Canada*, 2004.

[10] S. Turner and L. Chen, "Updated security considerations for the MD5 message-digest and the HMAC-MD5 algorithms," Internet Engineering Task Force (IETF), RFC-6151, 2011.

[11] T. Polk, L. Chen, S. Turner, and P. Hoffman, "Security considerations for the SHA-0 and SHA-1 message-digest algorithms," Internet Engineering Task Force (IETF), RFC-6194, 2011.

[12] D. Bider, "Use of RSA keys with SHA-256 and SHA-512 in the secure shell (SSH) protocol," Internet Engineering Task Force (IETF), RFC-8332, 2018.

[13] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.

[14] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," *Computer*, 2007.

[15] J. Zambreno, D. Nguyen, and A. Choudhary, "Exploring area/delay tradeoffs in an AES FPGA implementation," in *International Conference on Field Programmable Logic and Applications*, 2004.

[16] D. He and Z. Xue, "Multi-parallel architecture for MD5 implementations on FPGA with gigabit-level throughput," in *2010 International Symposium on Intelligence Information Processing and Trusted Computing*, 2010.

[17] J. He, H. Chen, and H. Huang, "A compatible SHA series design based on FPGA," in *ECTI-CON2010: The 2010 ECTI International Confernce on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, 2010.

[18] M. D. Rote, V. N, and D. Selvakumar, "High performance SHA-2 core using the round pipelined technique," in *2015 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2015.

[19] S. S. Omran and L. F. Jumma, "Design of multithreading SHA-1 and SHA-2 MIPS processor using FPGA," in *2017 8th International Conference on Information Technology (ICIT)*, 2017.

[20] S.-H. Lee and K.-W. Shin, "An efficient implementation of SHA processor including three hash algorithms (SHA-512, SHA-512/224, SHA-512/256)," in *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, 2018.

[21] H. L. Pham, T. H. Tran, V. T. Duong Le, and Y. Nakashima, "A high-efficiency FPGA-based multimode SHA-2 accelerator," *IEEE Access*, 2022.

[22] J.-P. Aumasson and D. J. Bernstein, "SipHash: a fast short-input PRF," in *International Conference on Cryptology in India*, 2012.

[23] C. Dobraunig, F. Mendel, and M. Schläffer, "Differential cryptanalysis of SipHash," in *International Conference on Selected Areas in Cryptography*, 2014.

[24] W. Xin, Y. Liu, B. Sun, and C. Li, "Improved cryptanalysis on SipHash," in *International Conference on Cryptology and Network Security*, 2019.

[25] L. He and H. Yu, "Cryptanalysis of reduced-round SipHash," *Cryptology ePrint Archive*, 2019.

[26] M. Olekšák and V. Miškovský, "Correlation power analysis of SipHash," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2022.

[27] C.-Y. Chu and M. Lukowiak, "Two step power attack on SHA-3 based MAC," in *2018 25th International Conference "Mixed Design of Integrated Circuits and System" (MIXDES)*, 2018.

[28] J. Alakuijala, B. Cox, and J. Wassenberg, "Fast keyed hash/pseudo-random function using SIMD multiply and permute," *arXiv preprint arXiv:1612.06257*, 2016.

[29] R. Elnaggar, R. Karri, and K. Chakrabarty, "Multi-tenant FPGA-based reconfigurable systems: Attacks and defenses," in *2019 Design, Automation Test in Europe Conference Exhibition*, 2019.

[30] "AXI-Stream Protocol Specification." [Online]. Available: https://developer.arm.com/documentation/ihi0051/b

[31] "AXI-Lite Protocol Specification." [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI4-Lite-Interface-Specification

[32] "An FPGA Implementation of SipHash." [Online]. Available: https://github.com/bwelte99/SipHash-FPGA-Accelerator

[33] "SipHash: High-speed pseudorandom function." [Online]. Available: https://github.com/veorq/SipHash

[34] *12th Generation Intel Core Processor Family Datasheet*, Intel Corporation, 2023, rev. 010.