Toward a New Methodology for Hardware Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study

Ekawat Homsirikamol and Kris Gaj
George Mason University, U.S.A.
Fairfax, Virginia 22030
email: {ehomsiri, kgaj}@gmu.edu

***Abstract*—The increasing number of candidates competing in cryptographic contests has made hardware benchmarking using the traditional Register-Transfer Level (RTL) methodology too inefficient and potentially unfair, especially at the early stages of the competitions. In this paper, we propose supplementing, and eventually replacing, this traditional RTL methodology with the use of High-Level Synthesis (HLS) tools. We apply our proposed HLS-based approach to FPGA benchmarking in the ongoing CAESAR contest, by comparing and ranking 16 authenticated ciphers, including the current standard, AES-GCM, and the primary variants of 13 Round 3 CAESAR candidates. After a careful survey of available HLS tools, we chose Xilinx Vivado HLS as our primary benchmarking tool. Our study has demonstrated high correlation between the rankings of the evaluated algorithms, obtained using both investigated methodologies. In particular, after applying HLS, the algorithm rankings in terms of two major performance metrics - throughput and throughput to area ratio - have either remained unchanged or have been affected only for algorithms with very similar RTL performance.**

***Index Terms*—authenticated ciphers, CAESAR, benchmarking, hardware, FPGA, HLS.**

I. Introduction & Motivation

Cryptographic contests have emerged as a commonly accepted way of developing cryptographic standards [1]–[3]. After candidates for a cryptographic standard are submitted to a contest, the evaluation process begins. The typical criteria are security, performance in software, performance in hardware, and flexibility. Although security is commonly accepted to be the most important criterion, it is rarely by itself sufficient to determine a winner. This is because multiple candidates generally offer sufficient security, and the trade-off between security and performance must be investigated.

Performance in software has been shown to be measured efficiently, accurately and reliably for a wide spectrum of processor types using the eBACS framework [4]. However, comparing performance (benchmarking) in hardware is still challenging, especially for a large number of candidates evaluated in the early rounds of cryptographic contests.

One possible way of addressing these difficulties is the distribution of the RTL development effort among multiple designers, working independently, as was the case for the Round 2 of the most-recent CAESAR Competition [5]. This approach has its advantages, such as a larger talent pool and smaller work effort for any particular team. However, it also has drawbacks. The designers can differ substantially in terms of experience and skill level. They can also devote substantially different amount of time and effort to the development and optimization of their code.

All the aforementioned problems can be potentially resolved by using high-level programming languages for formal specification, and related development tools for an automated high-level synthesis to hardware. Recent years have brought dramatic improvements in the quality, ease of use, and effectiveness of general-purpose high-level synthesis tools. In this paper, we investigate the use of C and the new generation of high-level synthesis tools, targeting FPGAs, for efficient benchmarking of CAESAR candidates in hardware.

In order to verify the potential validity of our approach, we have applied our proposed HLS-based methodology to the evaluation of 15 variants of 13 Round 3 CAESAR candidates, representing all single-pass candidate families, compared against each other and against the current NIST standard, AES-GCM.

Our primary goal is to demonstrate strong correlation between rankings obtained using both investigated methodologies. Our secondary goal is to show that the development time may be reduced significantly, and as a result all candidates can be implemented by the same group, or even a single designer (leading to a much fairer comparison). Additionally, even if the HLS-based approach is applied in parallel to the RTL-based evaluation, as it is the case during the current CAESAR competition, the HLS-generated results can be used to detect any suboptimal RTL code.

It should also be stressed that we do not advocate using HLS implementations as a replacement for RTL implementations. For maximum fairness, an HLS implementation should be compared with other HLS implementations only. The best time for such comparisons is at the early stages of cryptographic contests, when the number of candidates is large and the effort necessary to develop their RTL code not justified.

II. Previous Work

Previous studies have demonstrated that using HLS is a viable solution, at least in selected domains, such as digital signal and image processing [6]–[11]. The application of HLS

TABLE I: Benchmarking results, in number of clock cycles, after optimization, for four cryptographic benchmarks implemented using three academic HLS tools and one commercial HLS tool [15]. The best result obtained for a given benchmark using any of the investigated HLS tools is underlined. Due to the licensing restrictions, the developer of the commercial tool is not listed in [15].

Tools	Developer	aes-encrypt	aes-decrypt	sha	blowfish
Bambu [16]	Politecnico di Milano	1,485	2,585	51,399	57,590
DWARV [17]	Delft University of Technology	3,282	2,579	71,163	70,200
LegUp [18]	University of Toronto	1,191	4,847	81,786	64,480
Commercial	N/A	3,735	3,923	124,339	96,460
Manual	This Work	20	20	20,480	18,736
Best/Manual		60	129	2.5	3.1

to cryptography is relatively new, and has been reported in just a few earlier papers, such as [12]–[15].

The most comprehensive survey of modern HLS tools is the Oct. 2016 paper [15] by 12 leading experts in this field, representing three very active and prolific research centers at Delft University of Technology, the Netherlands, Politecnico di Milano, Italy, and University of Toronto, Canada. This survey covers a total of 12 commercial and 14 academic tools based on C, C++, or extended C, as well as 7 additional tools based on other languages. For four selected tools (3 academic and 1 commercial), listed in Table I, comprehensive performance tests were applied, using 17 benchmarks, representing multiple domains, such as communications, computer arithmetic, image processing, and cryptography. Four cryptographic benchmarks, most related to the topic of this paper, are described in Table II. They involve two block ciphers, AES and Blowfish, and one hash function SHA-1. The results are summarized in Table I.

This table demonstrates, that even if the best result obtained for a given benchmark using any of the four leading HLS tools is applied, the manual RTL approach still produces the results that are at least 2.5 times better in terms of the number of clock cycles alone. The differences in terms of the Execution time in microseconds, Throughput, and Throughput/Area ratio are even higher [15]. The penalty is particularly large, over 60, for benchmarks involving a single encryption/decryption (as in case of aes-encrypt and aes-decrypt), rather than processing of multiple consecutive blocks (as in case of sha and blowfish).

The previous study most similar to the topic of this paper concerns the comparison of HLS and RTL implementations of 5 final SHA-3 candidates [14]. However, authenticated ciphers are considerably different (and more complicated) than hash functions. The good correlation of results for one class of algorithms does not imply the similar correlation for another class. Additionally, any previous studies were much smaller in scope and concerned only past competitions.

III. Choice of HLS Tool

A commercial tool, Vivado HLS, was chosen because it is supported by the largest FPGA company - Xilinx, and at the same time is easily affordable to researchers in both academia and industry. It supports the larger number of performance optimizations than leading academic tools [15] [21]. It has very

TABLE II: Description of cryptographic benchmarks from the CHStone Benchmark Program Suite for Practical C-based High-Level Synthesis [15], [19], [20]

Benchmark	Description
aes-encrypt	AES Key Scheduling + AES Encryption of **1** 128-bit block
aes-decrypt	AES Key Scheduling + AES Decryption of **1** 128-bit block
sha	Hashing of **256** 512-bit blocks using SHA-1
blowfish	Blowfish Key scheduling + Blowfish Encryption of **650** 64-bit blocks in the CFB64 mode

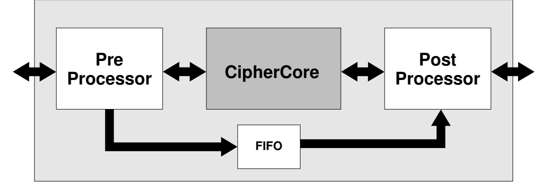

Fig. 1: Partition of a High-Speed AEAD Core into major units

good documentation [21] and user support. It also produces portable HDL source code that is platform agnostic. *Our study was limited to Xilinx devices only because of the current licensing restrictions of Vivado HLS, which prohibit its use targeting other FPGA families (such as Altera) or ASICs*. Other HLS tools targeting specifically ASIC technology exist, but are typically expensive and thus beyond the reach of academic groups. We believe that using multiple HLS tools, including academic tools with limited documentation and support, would be too time-consuming, and would detract from the focus of our study, which is the comparison of algorithms and design methodologies, and not the comparison of the available HLS tools.

IV. Design Methodology

A. RTL and HLS-based Design Flows

The detailed RTL-based design methodology for developing high-speed implementations of authenticated ciphers compliant with the CAESAR API [22] is described in [23]. According to this methodology, the entire AEAD (Authenticated Encryption with Associated Data) core is first divided into four major units, shown in Fig. 1, namely: *PreProcessor*, *PostProcessor*, *FIFO*, and *CipherCore*. The first three of these units can be shared by high-speed implementations of all authenticated ciphers, as long as these implementations are compliant with [22].

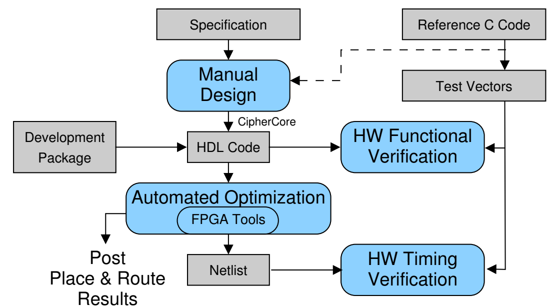

Fig. 2: RTL-based development and benchmarking flow

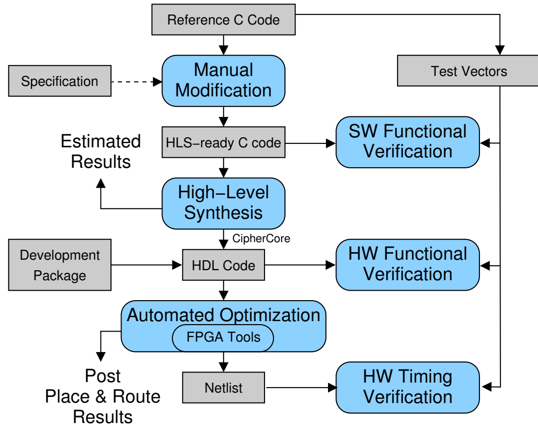

Fig. 3: HLS-based development and benchmarking flow

A traditional, RTL-based hardware benchmarking flow, shown in Fig. 2, typically starts with the translation of an informal specification to a hardware description language (HDL) code, e.g., the RTL code written in VHDL or Verilog. In the RTL design of authenticated ciphers [23], this translation involves developing a block diagram of the Datapath and an Algorithmic State Machine (ASM) chart of the Controller. Both parts of the design are constrained by the CipherCore interface, described in [23]. They are also influenced by the optimization target, common for all investigated algorithms, which is typically the maximum Throughput/Area ratio.

After the manual design of the CipherCore is completed, this code is combined with the HDL code of other units, extracted from the Development Package for Hardware Implementations Compliant with the CAESAR Hardware API [24], and verified using functional simulation of HDL code. This verification is based on test vectors, generated using Reference C Code of a given algorithm, and takes advantage of a generic testbench provided as a part of the Development Package.

As a next step, the post-place & route results and the final netlist are generated. Finally, the obtained netlist is undergoing a hardware timing verification for the last check up.

The proposed HLS-based development and benchmarking approach is shown in Fig. 3. For the maximum efficiency and simplicity of the development flow, only CipherCore is modeled in HLS-ready C code. The remaining three units, *PreProcessor*, *FIFO*, and *PostProcessor*, are extracted from the Development Package, and used without any changes in both RTL-based and HLS-based design flow.

The HLS-ready C code of CipherCore is obtained by the manual modification of the Reference C code of a given algorithm. These modifications include a) converting the Application Programming Interface, to infer the interface of the CipherCore, specified in detail in [23], b) adding the HLS tool directives, provided as pragmas, to minimize the number of clock cycles per input block and share FPGA resources, c) code refactoring, which involves modifying the code to make it more suitable for parallelization and resource reuse, and d) combining all modes of operation into a single underlying function for resource reuse.

Once these modifications are completed, the code is verified in software for the correct functionality. Afterward, the HLS-ready C code is processed by the HLS tool to generate an RTL HDL code and estimated results. If the obtained results are worse than expected, e.g., the number of clock cycles per block is higher than the number of cipher rounds plus a small constant (to be expected for the basic iterative architecture), or the resource utilization is much higher than expected, e.g., one observes very high register count, the HLS-ready code is modified, and the entire process repeated. If the HLS-ready C code of CipherCore performs as expected, this code is used to generate the equivalent RTL code. The obtained RTL code is further combined with the RTL code of other units, and used to generate the final post-place and route results, as well as the final netlist, verified using timing simulation.

B. Inferring the CipherCore Interface in C/C++

One of the most important design tasks is the conversion of the software API of CAESAR candidates, specified at [3], in such a way that the hardware interface of the CipherCore, specified in [23], is automatically inferred by the synthesis tool. This conversion is done as follows.

In the first step, input and output ports of CipherCore are divided into groups that rely on the same handshaking signals, as shown in Fig. 4. These groups are called *Public*, *Secret*, *Key_Ctrl*, *Operation*, *Output*, and *Tag_Verification*. Each group is represented in C using a structure with appropriate fields, corresponding to individual buses or signals.

Three structure types, defined in Fig. 4, *public_bus*, *secret_bus*, and *output_bus*, are then declared in Fig. 5 as streaming interfaces. As a result, the handshaking signals *valid* and *ready*, controlling the ports of the respective groups, are inferred automatically. Similarly, for the non-streaming output port, *msg_auth_valid*, the control signal *msg_auth_done* is inferred automatically. Static inputs that remains constant for the entire duration of the function call, such as *decrypt*, can be represented using basic non-streaming data types, such as *bool*, and require no control signals. However, inputs that may get updated in the middle of the function call, such as *key_update*, should be declared using the *volatile* keyword, to ensure that the tool correctly synthesizes the desired interface.

Each field of the structures *public_bus*, *secret_bus*, and *output_bus* can have a either a standard type, such as *bool*, or a custom type, such as *data_t*, *data_bytes_t*, *bsize_t*, and *secret_t*. These custom types are defined in Fig. 6. The *ap_uint<WIDTH>* type is a special type provided by Vivado HLS libraries, representing an unsigned integer of arbitrary width, given by the parameter *<WIDTH>*. For example, *ap_uint<BLOCKSIZE>*, with *BLOCKSIZE* defined as a constant equal to 128, represents a 128-bit unsigned integer. Thus, the corresponding port, inferred by an input or output of this type is a 128-bit bus.

Once this template is created, a programmer can easily change any port width to the desired value by simply changing

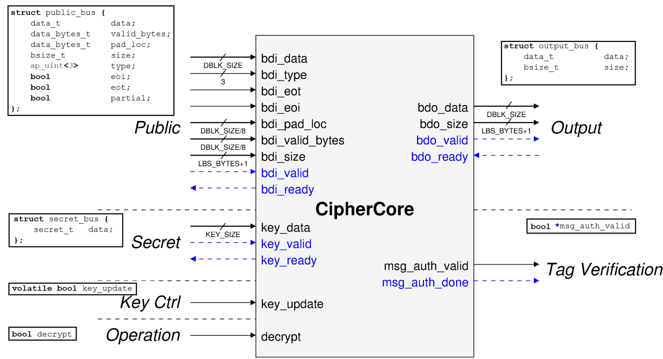

Fig. 4: Translation of the Cipher Core interface to the respective structure types in C

```
typedef hls::stream<public_bus> BDI_STREAM;
typedef hls::stream<secret_bus> KEY_STREAM;
typedef hls::stream<output_bus> BDO_STREAM;

void hls_ciphercore (
    // Post-Processor
    BDO_STREAM& bdo_bus,
    bool *msg_auth_valid,
    // Pre-Processor
    BDI_STREAM& bdi_bus,
    KEY_STREAM& key_bus,
    bool decrypt,
    volatile bool key_update
);
```

Fig. 5: Cipher Core interface described in C

```
/* Interface parameters */
#define BLOCKSIZE 128
#define KEYBUS 32
#define LBS_BYTES 4

/* Interface data types */
typedef ap_uint<KEYBUS>       secret_t;
typedef ap_uint<BLOCKSIZE>    data_t;
typedef ap_uint<BLOCKSIZE/8> data_bytes_t;
typedef ap_uint<LBS_BYTES+1> bsize_t;
```

Fig. 6: Definition of the Cipher Core bus widths in C

TABLE III: Comparison of Reference C vs. HLS-ready C/C++

Data	Reference C	HLS-ready C/C++
Access	Random: Data can be accessed at any location multiple times	Serial: Previously accessed data must be maintained inside of the code if required
Width	Byte/Word	Block size
Total Size	Known in advance	Unknown until read as a part of input
Status	Always available	Availability unknown until the time of read

values of constants *BLOCKSIZE*, *KEYBUS*, and *LBS_BYTES*, shown in Fig. 6.

Replacing a standard software API of Reference C code with the discussed above streaming interface of HLS-ready C has further substantial consequences on the entire code. The major differences between the two implementations are summarized in Table III.

C. HLS C Optimization Methodology

This section describes our HLS C optimization methodology. The aim of this optimization is to infer an HLS-generated RTL code that has performance the same or better than a manually developed RTL code. The first step involves creating an HLS-ready baseline implementation from the reference C code. At this stage, any variables and buffers that rely on dynamic allocation are replaced by their static versions.

Function Reuse can be considered as one of the most critical factors in achieving an optimized design that has performance comparable to RTL-based design. In software, a repetition of a function call does not drastically affect performance or memory utilization of any program. However, unless an appropriate action is taken, a C synthesizer can treat each function call as a separate instantiation of a hardware module. As a result, multiple function calls can significantly increase the design area and critical path.

One of the approaches to mitigate this effect is to add a limit on the resource allocation for a specific function, in order to limit its reuse. In Vivado HLS, the ***RESOURCE*** pragma facilitates this process. Rewriting source code in order to minimize the number of function calls is an alternative approach. While this approach requires more effort, it allows a finer control of the design, which can help to produce a more consistent results for benchmarking purposes.

Data storage. The next step is optimizing data types and sizes of arrays, and providing the tool with directives on how these arrays should be translated into hardware. Vivado HLS synthesis tool supports a flexible base data type, ***uint[size]***. Utilizing a correct size for our needs can significantly increase design efficiency.

The speed at which data can be accessed is determined by its type and/or array dimension. By default, one clock cycle is required to access an element of an array (memory). This feature can significantly limit the design's throughput. Flattening the memory can alleviate this problem. This, however, can produce an adverse effect in terms of resource utilization. Thus, memory partitioning should be done very carefully, and should use the overall design's structure as a guide.

A memory can either be a ROM or RAM depending on the pragma specified. In Vivado HLS, a ROM inference can be done via the ***RESOURCE*** pragma, using an asynchronous ROM (ROM_1P_1S) core, which is one of many hardware types that can be specified. This type is particularly useful when the designer does not want the tool to implement a given storage component using BlockRAM. The change to asynchronous ROM can significantly reduce the design latency.

Loop Optimization. In order to fully realize the potential of the design after data storage has been optimized, all loops of the program must be optimized as well. In Vivado HLS, the ***UNROLL*** pragma is the best directive to realize this potential. This directive informs the tool to unroll the operations within a loop, so they can be executed in parallel, thereby increasing the overall throughput of the loop. However, an additional code refactoring, based on the Speculation and Loop Invariant techniques can be still required [21].

The final aspect of code optimization, and perhaps, one of the most difficult ones is temporal parallelism extraction, which is essentially pipelining. While the tool can infer pipelining for standard DSP operations, it is not yet sophisticated enough to facilitate a more complex pipelining

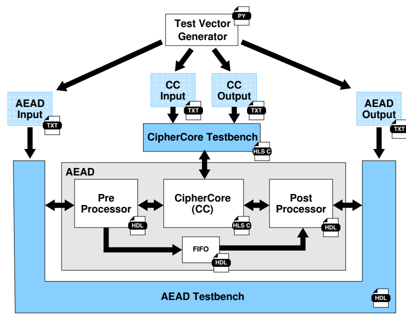

Fig. 7: Verification framework. Notation: PY - Python app, TXT - ASCII Text File, HDL - Hardware Description Language code, AEAD - Authenticate Encryption with Associated Data (name of the top-level core).

behavior, e.g. block-level pipelining, required by cryptographic algorithms. As a result, careful code refactoring has to be performed in order to guide the tool to infer a correct operation.

D. Verification and Debugging

Fig. 7 presents our entire verification and debugging framework. This framework relies on a generic test vector generator, available as a part of [24]. This generator can automatically create the equivalent test vectors for both software and hardware verification. Only when the HLS-ready C code is judged to be fully operational based on software functional verification, the corresponding CipherCore HDL code is generated using HLS, as shown in Fig. 3. The designer then proceeds to the hardware functional verification, which is performed using exactly the same approach as in the RTL development flow. If the entire HDL code, including the HLS-generated CipherCore passes this hardware verification, this code is then used to generate the final post-place and route results.

V. Features of the Implemented Algorithms and their Hardware Architectures

In order to verify our hypothesis, 16 authenticated ciphers, representing 13 candidate families and the current FIPS standard, AES-GCM, have been selected as our test case. All investigated candidate families have been qualified to Round 3 of the competition, which has been in progress at the time of the submission of this paper. The only two families not represented in this study are: 1) AEZ - as this family does not have a single-pass representative, and implementing a two pass-authenticated cipher was beyond the scope of this study, and 2) Keyak, whose RTL implementation (developed by the authors of the algorithm) is not fully compliant with the CAESAR Hardware API due to its design novelty. For all candidates, an effort was made to implement at least the primary variant, targeting Use Case 2: High-performance applications.

Taking into account that the majority of the RTL implementations available at the time of writing were implementations of the Round 2 variants of the Round 3 candidates (without Round 3 tweaks), for the fairness of comparison, the same exact variants were used for our HLS implementations, as shown in Table IV.

The major features of the implemented variants of authenticated ciphers and their RTL HDL and HLS C designs are summarized in Table IV. The implemented ciphers represent three major classes of authenticated ciphers: block-cipher-based (mostly, but not exclusively, built around the AES Primitive), stream-cipher-based, and the sponge constructions [3].

All RTL implementations used in this study have been developed as a part of the open Round 2 VHDL/Verilog development/benchmarking process, summarized in [5][1] The primary contribution of this work is the development of HLS-ready implementations of all investigated candidates, and then the comparison of the obtained results, with the results reported for the existing RTL implementations in the on-line database of results [25], at the time of the paper submission.

Similarly to the open CAESAR Round 2 VHDL/Verilog development process, our primary optimization target is the best Throughput to Area ratio, where Throughput is defined as Throughput for long messages, and Area is expressed in terms of the number of LUTs of Xilinx FPGAs.

In all algorithms investigated in this paper, the speed of encryption and decryption is the same. Additionally, the Associated Data processing speed is at least as high as the speed of encryption. The formula for the throughput for long messages is given by (1).

$$Throughput = Block_Size/(Cycles_per_block \cdot T_{clk}) \quad (1)$$

The number of clock cycles per one block of message/-ciphertext (Cycles_per_Block) is summarized in Table IV. This number is similar, but not necessarily the same for the RTL-based and HLS-based approach. For the majority of implemented ciphers, the automatically generated Controller of the CipherCore (initially described in C) tends to be sub-optimal, compared to the manual design using RTL VHDL. This is because the HLS-generated unit is often not capable of supporting an overlap between the completion of the last round of encryption/decryption and reading the next input block. Moreover, additional clock cycles are often added by the HLS scheduler when entering/exiting a control region. As a result, the throughput of the HLS-based design tends to be lower than the throughput of the manual design, even if they both operate at the same clock frequency.

VI. Results and Discussion

A. Benchmarking

The FPGA families targeted in this study, Virtex-6 and Virtex-7, are a subset of families used in the comprehensive CAESAR Round 2 benchmarking study [5]. Exactly the same FPGA devices have been used as target devices in this project.

In order to minimize any discrepancies between the rankings obtained using HLS and RTL-based approaches, we applied the same tools and optimization techniques to both manually

[1] We do not claim any authorship of any subset of these implementations, as doing so, would violate the anonymity of this submission.

TABLE IV: Features of the implemented variants of authenticated ciphers and their hardware designs. Notation: AES[1] - one-round AES, LFSR - Linear Feedback Shift Register, LRX - Logic-Rotation-XOR, SPN - Substitution-Permutation Network.

Algorithm	Variant / SW Reference	Basic Primitive	Block Size	Rounds	Cycles_per_block RTL	Cycles_per_block HLS	Ratio HLS/RTL
Block-cipher-based							
AES-GCM	aes128gcmv1	AES	128	10	11	12	1.09
AEGIS	aegis128l	AES[1]	256	1	1	1	1.00
AES-OTR	aes128otrpv2/v3[a]	AES	128	10	12	12	1.00
CLOC-AES	aes128n12t8clocv2	AES	128	10	11	12	1.09
COLM	colm0v1	AES	128	10	11	12	1.09
Deoxys	deoxysneq128128v13	Deoxys-BC	128	14	15	16	1.07
JAMBU-AES	aesjambuv2	AES	128	10	10	12	1.20
JAMBU-SIMON	simonjambu96v2	SIMON	48	52	54	55	1.02
OCB	aeadaes128ocbtaglen128v1	AES	128	10	12	12	1.00
SILC-AES	aes128n12t8silcv2	AES	128	10	10	12	1.20
Tiaoxin	tiaoxinv2	AES[1]	128	1	1	1	1.00
Stream-cipher-based							
ACORN	acorn128v2	LFSR	32	1	1	1	1.00
MORUS	morus1280128v1	LRX	256	1	1	1	1.00
Sponge-construction							
Ascon	ascon128v11	SPN	128	8	9	10	1.11
Ketje	ketjesrv1	Keccak-f	32	1	1	1	1.00
NORX	norx3241v2	LRX	384	4	4	6	1.50

[a] v2 used in the HLS implementation, v3 in the RTL implementation.

written and automatically generated HDL code. The synthesis of HDL code was accomplished using Vivado HLS v2015.4, with VHDL as a target language. Logic synthesis and implementation (mapping, placing, and routing) were performed by ISE Design Suite v14.7 for Virtex 6, and by Xilinx Vivado v2015.4 for Virtex 7. In terms of optimization of tool options, for Virtex 6, we have used Xilinx ISE and ATHENa [26], for Virtex 7, we have applied 25 default optimization strategies available in Xilinx Vivado.

The options of synthesis and optimization tools have been set to infer the use of zero Block RAMs and zero DSP units inside of *AEAD*. These FPGA-specific resources were not used because otherwise area (resource utilization) would become a vector, such as (#LUTs, #DSP units, #BRAMs). Additionally, a previous study for similar cryptographic algorithms has demonstrated that area in logic resources (such as LUTs) in FPGA technology correlates well with area in gate equivalents in ASIC technology [27]. As a result, using LUTs alone as a unit of area seems to give us an additional insight regarding the relative cost of implementing investigated ciphers in ASICs.

B. Analysis of Results

In terms of the ratio of the number of clock cycles required to process data by circuits generated using HLS-based and RTL-based approaches, our HLS C implementations significantly outperform the earlier attempts at cryptographic benchmarks, reported in [15], and shown in Table I. The smallest ratio reported in Table I is 2.5 for the *sha* benchmark. On the other hand, in our implementations, even the worst ratio, reported in the rightmost column of Table IV is significantly smaller, at 1.50, and all remaining ratios are at or below 1.20. Additionally, for all algorithms using one clock cycle per block in RTL, this ratio is equal to an ideal value of 1.00. Such big difference in these results can be partially explained by the fact that the authors of [15] have not seriously attempted to match the performance of manual designs, and might have been even unaware how many clock cycles these manual designs required to perform the same task. Still, we believe that beating these previous attempts by such a large margin, and accomplishing the equal number of clock cycles for 6 out of 16 designs, and comparable (smaller than or equal to 20%) for 15 out of 16 algorithms is a significant achievement by itself.

In Figs. 8 and 9, we show the rankings of all implemented ciphers in terms of Throughput and Throughput to Area ratio, respectively. For each of the two FPGA families, Virtex-6 and Virtex-7, these graphs show absolute values of the respective metrics for both RTL and HLS-based implementations. In the legends of these diagrams, the pairs (RTL_position, HLS_position) show the relative rankings of each candidate obtained using the respective design flow. For all three performance metrics, a substantial percentage of relative positions remain stable, independently of the benchmarking methodology and the target platform. At the same time, some changes in the relative rankings are clearly unavoidable, especially for the sub-groups of implemented algorithms with a relatively similar performance in RTL according to a particular performance metric. For example, in Fig. 8, the leading three candidates, AEGIS, Tiaoxin, and MORUS, are relatively close to each other in RTL for both Virtex-6 and Virtex-7. However, for HLS on Virtex-6, their relative ranking changes, while for Virtex-7 it remains the same. This diagram also reveals that six algorithms - AEGIS, Tiaoxin, MORUS, NORX, ACORN, and Ascon, consistently outperform AES-GCM, independently of the applied design flow.

In Figs. 10 and 11, we present the ratios of RTL to HLS results for Xilinx Virtex-6 in terms of Throughput and Throughput/Area ratio, respectively. In both cases, the desired range, marked in green, is from 1.3 down to 0.9. In general, the smaller the spread of ratios, the better the chances that the change from RTL to HLS will not modify the rankings.

In terms of Throughput, the only outliers are SILC, Ketje,

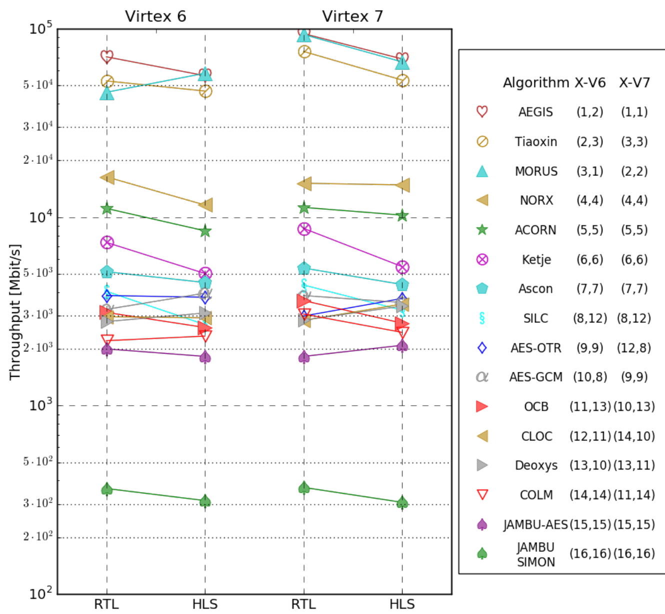

Fig. 8: Algorithm rankings based on Throughput

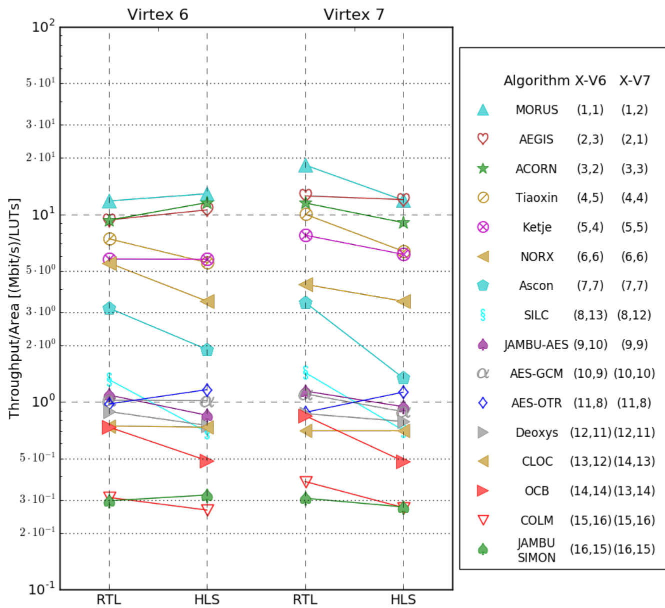

Fig. 9: Algorithm rankings based on Throughput to Area Ratio

NORX, and ACORN, which have suboptimal HLS code in terms of this performance metric. On the other hand, MORUS and AES-GCM have slightly suboptimal RTL code. All other algorithms fall within the desired range.

In terms of Throughput/Area, six algorithms, SILC, Ascon, NORX, OCB, and Tiaoxin, have somewhat inefficient HLS implementations. These implementations can be potentially still improved through the code refactoring. However, the improvement is particularly difficult for NORX, where the majority of the advantage of the RTL code comes from the smaller number of Cycles_per_Block, 4 for RTL, and 6 (50% more) for HLS, as shown in Table IV. On the other hand, three algorithms, ACORN, AES-OTR, and AEGIS, have suboptimal RTL code in terms of Throughput/Area.

VII. Conclusions and Future Work

The design methodology based on high-level synthesis can potentially enable hardware benchmarking at the early stages of cryptographic competitions, when the number of candidates is still large (e.g., over 50 in the last two contests, SHA-3 and CAESAR) [2], [3]. Whenever multiple designers are involved, there is always a risk that the observed differences come from their skill level, rather than actual algorithm differences. Using HLS can mitigate this risk by allowing a single designer to produce implementations of multiple (and even all) candidates.

The correct inference of desired architecture may require substantial changes in the reference C code. During this process, the programmer needs to repeatedly check the estimated results, using the design cycle provided by the HLS tool, to ascertain whether the inferred circuit corresponds to the desired hardware architecture (in this study, the basic iterative architecture). This check can be made by determining the number of clock cycles per block, which should be closely related to the number of the cipher rounds (typically just the number of cipher rounds + a small constant, such as 1 or 2). An initial HLS run gives instead thousands of clock

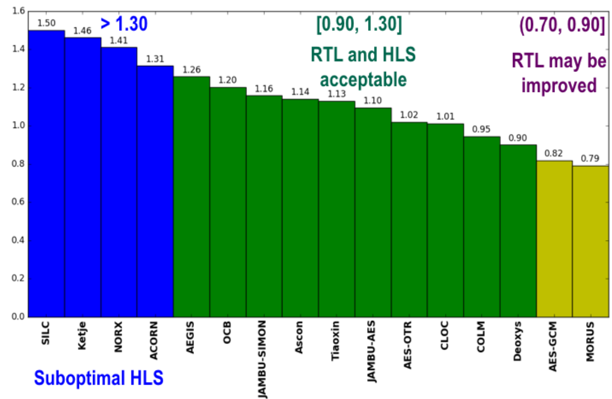

Fig. 10: RTL to HLS Ratios for Throughput in Virtex-6

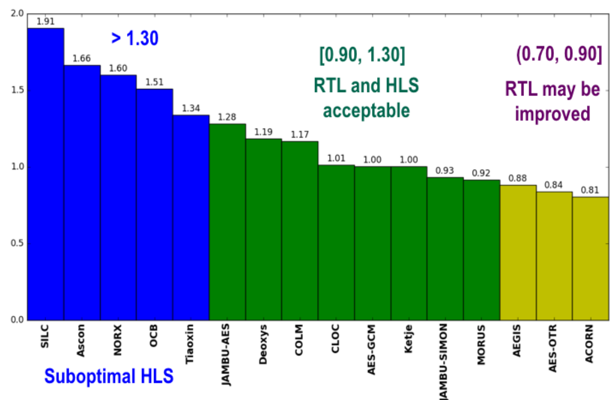

Fig. 11: RTL to HLS Ratios for Throughput/Area in Virtex-6

cycles per block [13]. Thus, any modifications to the code may safely stop when the number of clock cycles per block reaches the desired target. This feature is very specific to high-speed implementations of symmetric-key cryptographic algorithms (such as symmetric block ciphers, hash functions, and authenticated ciphers). It may not apply to other domains, or even to implementations of public-key cryptography.

In the HLS design approach, the user focuses on the functionality of the design (datapath); the control logic is inferred. Additionally, a C testbench development and code verification is also much easier to conduct. These two major factors has led to the estimated speed-up by a factor of 3 to10 in terms of the development time. The exact value of this speed-up has been dependent on multiple factors, such as the algorithm itself, the quality of the specification and the reference implementation in C, as well as relative skills of the respective designers.

The HLS approach clearly identified that seven candidates - MORUS, AEGIS, ACORN, Tiaoxin, Ketje, NORX, and Ascon - consistently outperformed the current standard AES-GCM in terms of the throughput to area ratio. However, two other candidates SILC and JAMBU-AES swapped positions with AES-GCM (at least for one FPGA family), depending on the implementation approach.

Our case study, based on 16 modern authenticated ciphers, has demonstrated strong (but not ideal) correlation in terms of the algorithm rankings according to two major performance metrics for high-speed implementations: throughput and throughput to area ratio. While the approach we have used requires more effort compared to the traditional HLS design approach described in [15], the quality of results is substantially better and comparable to the manual RTL approach, while time savings remain substantive.

Future work will involve 1) development of the step-by-step designer's guide, including general strategies for the C code refactoring and pragma insertion; 2) release of the full source code of all HLS-ready C implementations developed as a part of this study, to be used as comprehensive examples by other designers, and as benchmarks by developers of HLS tools; 3) design space exploration of the authenticated cipher implementations through the inference of selected folded, unrolled, and pipelined architectures in the HLS-ready C code; 4) experiments with multiple HLS tools, including automated translation of Vivado HLS pragmas to the pragmas of leading academic tools: Bambu, DWARV, and LegUp; 5) identifying inherent limitations of the current generation of HLS tools and proposing possible improvements and research avenues.

REFERENCES

[1] National Institute of Standards and Technology. (2000, Oct.) Report on the Development of the Advanced Encryption Standard (AES). [Online]. Available: http://csrc.nist.gov/archive/aes/round2/r2report.pdf

[2] ——. (2012, Nov) Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf

[3] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. Cryptographic competitions. [Online]. Available: http://competitions.cr.yp.to

[4] D. J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. [Online]. Available: http://bench.cr.yp.to

[5] E. Homsirikamol et al. (2016, Sep) Toward Fair and Comprehensive Benchmarking of CAESAR Candidates in Hardware. [Online]. Available: http://www.nuee.nagoya-u.ac.jp/labs/tiwata/diac2016/slides/diac2016_08_Kris.pdf

[6] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[7] Berkeley Design Technology, Inc. (2010) High-Level Synthesis Tools for Xilinx FPGAs. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/Xilinx_hlstcp.pdf

[8] J. Cong et al., "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[9] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Oct 2011, pp. 1102–1105.

[10] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level Synthesis: Productivity, Performance, and Software Constraints," *Journal of Electrical and Computer Engineering*, Jan 2012.

[11] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An Overview of Today's High-level Synthesis Tools," *Des. Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, Sep. 2012.

[12] S. Morioka, T. Isshiki, S. Obana, Y. Nakamura, and K. Sako, "Flexible architecture optimization and ASIC implementation of group signature algorithm using a customized HLS methodology," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2011, pp. 57–62.

[13] E. Homsirikamol and K. Gaj, "Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study," in *Reconfigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014.

[14] ——, "Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study," in *11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings Series*, Apr 2015.

[15] R. Nane et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[16] P. di Milano, *PandA*, Mar 2017. [Online]. Available: https://panda.dei.polimi.it

[17] R. Nane et al., "Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 619–622.

[18] U. of Toronto, *LegUp High-Level Synthesis*, Mar 2017. [Online]. Available: http://legup.eecg.utoronto.ca

[19] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. CHStone: A Suite of Benchmark Programs for C-based High-Level Synthesis. [Online]. Available: http://www.ertl.jp/chstone/

[20] ——, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[21] Xilinx. (2017, June) Vivado Design Suite User's Guide: High-Level Synthesis.

[22] E. Homsirikamol et al., "CAESAR Hardware API," Cryptology ePrint Archive, Report 2016/626, June 2016, http://eprint.iacr.org/.

[23] ——. (2017) Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=CAESAR

[24] ——. (2017) Development Package for Hardware Implementations Compliant with the CAESAR Hardware API. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=CAESAR

[25] CERG. (2017) Database of FPGA Results for Authenticated Ciphers. [Online]. Available: https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view

[26] K. Gaj et al., "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 414–421.

[27] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs," Cryptology ePrint Archive, Report 2012/368, 2012.