Abstract—Recently, “high-level synthesis” tools for generation of FPGA hardware based on descriptions of algorithms in high-level languages have come into the mainstream. These tools can likely reduce time-to-market by automating the process of mapping an algorithm into hardware, but do they outperform traditional implementation techniques for cryptographic applications? This project will attempt to implement AES and the AES-GCM authenticated cipher and AES-128 block cipher using high-level synthesis frameworks, and characterize the realized FPGA implementations in terms of metrics such as throughput-to-area ratio.

Index Terms—Ciphers, Cryptography, Encryption, Field programmable gate arrays, High Level Synthesis, Message authentication.

I. INTRODUCTION

IMPLEMENTATION of digital systems on programmable logic devices such as FPGAs has traditionally required developing an RTL description in a hardware description language such as VHDL or Verilog. This process can be tedious and presents many challenges: Low-level descriptions of the hardware can become quite complex. Formal verification of HDL code through RTL simulation can be prohibitively time-consuming. Furthermore, low-level HDL code describes a specific set of architectural decisions in a design, which may not be optimal.

“High-Level Synthesis” ("HLS") tools such as Xilinx's Vivado HLS and The University of Toronto's LegUp allow synthesis of FPGA hardware from descriptions written in C or C++, and validation “testbenches” can be developed in these languages as well. The Cryptol language (a domain-specific language developed by Galois, Inc.) has gained traction in the cryptological community by allowing cryptographic algorithms to be described in a way that is both natural to cryptographers and precise enough to execute on a computer. Furthermore, Cryptol code can be compiled into synthesizable HDL code.

In this project, the AES-GCM (the “Galois/Counter Mode” authenticated cipher [1]) is implemented using Vivado HLS, and AES-128 under Cryptol. Where possible, synthesis parameters were provided to the HLS tools to attempt to maximize throughput while minimizing area.

II. PRIOR WORK AND APPROACH

In [2], AES and several “modes” of AES were implemented using high-level synthesis. This project builds on those findings to attempt to implement the AES-GCM authenticated cipher with a 128-bit key, as described in [1].

A. Circuit Interface

The circuit interface depicted in is inspired by [4], and refined in [5] in an attempt to make this implementation more directly comparable to other implementations of authenticated ciphers.

Figure 1: Top-level interface for AES-GCM

Input and output data transfers are governed by simple handshaking signals. Two input ports are provided – the first ("pdi") for “public” data inputs such as initialization vectors, message data, and additional authenticated data (AAD), and the second ("sdi") for “secret” data inputs such as the encryption key. Data is passed to the cipher as a series of structured “segments” and “instructions”, which will appear on one of the data ports as a series of 128-bit words. At a high-level, these would include the following:

Instruction word:
1. Use Authenticated Encryption Mode
2. Use Authenticated Decryption Mode
3. Load Key

Segment Header:
Specifies the type/length of segment to follow:
1. Initialization Vector
2. Associated Data
3. Plaintext (for Encryption)
4. Ciphertext (for decryption)
5. Tag (for verification during decryption)
6. Key
The segment header also contains a “message ID” field which is used as a sanity check to ensure that each segment relates to the same encryption/decryption operation.

**Segment Body:**
Carries the data associated with the type specified in the Segment Header.

In the encryption mode, when all plaintext and AAD has been processed, the 128-bit tag is presented on the output port until the downstream circuit is ready to receive it.

In the decryption mode, a 128-bit “result word” is presented at the output port indicating whether the tag was valid. Additional encryption and decryption operations cannot occur until the downstream circuit accepts the output.

If unexpected/invalid input is encountered, the “error” signal is asserted, and a code describing the type of error appears on the “ecode” bus.

### III. HIGH-LEVEL DESIGNS

In this section, a brief introduction to AES-GCM is given to provide context for the following description of the C and Cryptol implementations.

#### A. Galois/Counter Mode

Detailed in [1], the “Galois/Counter Mode” (GCM) is an authenticated cipher which, while similar in concept to the “counter” (CTR) block cipher mode of operation, includes an additional “hash” component composed of a multiplication by a key-derived element of the Galois field GF($2^{128}$) , which is used to generate the authentication tag. A particular block cipher (in this project, 128-bit AES) is used to produce both the ciphertext and the “H” factor used in the “GHASH” hashing function.

Figure 2 depicts a high-level block diagram of GCM. The blocks labeled $E_K$ denote the application of the forward block cipher under the key K. The blocks labeled $\text{mult}_H$ denote multiplication by H in GF($2^{128}$) with the irreducible polynomial stated in [1].

Broadly speaking, the mode consists of two main functions – “GCTR”, which corresponds to the applications of the block cipher and ensuing XOR operation outlined in the top of Figure 2, and “GHASH” which corresponds to the Galois multiplications by H outlined in the lower half of Figure 2. The “H” factor is derived from the key and is defined as the ciphertext produced by the application of the block cipher to an input of all zeros.

Like the CTR mode of operation, GCM’s decryption mode has a very similar structure to the encryption mode, allowing the same hardware (with minor accommodations) to be used for both encryption and decryption. In particular, only the forward cipher is required, the reverse cipher is not used under GCM.

#### B. C Implementation of AES-GCM

For synthesis under Vivado HLS and LegUp, a C implementation of AES-GCM was developed. To comply with the interface described in Figure 1, the top-level function `aes_gcm()` was declared as:

```c
void aes_gcm(
    /* Public data port */
    pdi_t  * pdi,
    /* Secret data port */
    sdi_t  * sdi,
    /* Output data ports */
    dout_t  * dout,
    word8   * ecode
);
```

**Figure 3: C Top-level Interface**

The `pdi_t`, `sdi_t`, and `dout_t` types are typedef's to a 128-bit integer type. Since the input and output ports in Figure 1 are designed to interface to external FIFOs, pointers are used to access inputs and generate outputs – the act of dereferencing a pointer from C corresponds to a read operation on an input FIFO and the act of dereferenced assignment in C corresponds to a write operation on an output FIFO.

Like in an RTL-based hardware design, the `aes_gcm()` function has an internal state, which is represented by variables declared with the C “static” keyword. While it can be considered poor practice to maintain state within a function between calls when developing C for software, it's required in order to maintain state between invocations of a function under HLS – this is one of many peculiarities that arise when writing C code for HLS. When implemented in hardware, all
HLS designs have some form of compulsory “reset” signal which resets the state of any static variables to the initialization values given in the C code (or, as required by the C language, to 0 if no initialization is given).

Encryption and decryption are accomplished by passing a sequence of “instructions” to the circuit, each of which is 128 bits wide. In C, this corresponds to one invocation of the toplevel function. The actions taken by the aes_gcm() function on a particular invocation are determined by its internal state. Typically, on each invocation, aes_gcm() will read from either its “public” (pdi) port or “secret” (sdi) port, and then perform some operation on the obtained data, updating its internal state and optionally writing to its “output” (dout) port. If an erroneous instruction is received, aes_gcm() writes an error code to its “ecode” output port.

The order in which the GCTR and GHASH operations are to be applied to message data is effectively forced by the order of operations defined in the GCM standard – the key must first be provided and H derived, then the initialization vector for the GCTR function, followed by the AAD and finally the plaintext (for encryption) or ciphertext (for decryption).

Figure 4 describes the state transitions of aes_gcm() throughout the authenticated encryption/decryption process. Note that the process is similar both for encryption and decryption. Initially, the key must be provided. The key is stored internally and H is generated. The circuit then waits in the IDLE_KEYED state until either asked to re-key or begin an authenticated encryption or decryption operation. If either of the latter, the encryption/decryption mode is recorded internally and then the initialization vector, AAD, and message (either plaintext or ciphertext) are read in-turn. When in the AAD_WAIT state, all AAD data received at the input is echoed to the output, and when in the MSG_WAIT state, the ciphertext is sent to the output when operating in encryption mode, or the plaintext is sent to the output when operating in the decryption mode.

Figure 4: C AES-GCM State Diagram
1 AES Implementation
The C implementation uses an existing implementation of AES which is suitable for HLS, kindly provided by the authors of [2]. The forward cipher is implemented in C by the function AES_encrypt() – taking a 128-bit input, 128-bit key, and produces a 128 bit output. When implemented in isolation in Vivado HLS with the appropriate HLS directives specified, this function can be realized in 10 cycles per encryption.

2 GCTR Implementation
The GCTR portion of GCM consists primarily of the application of the block cipher (AES) to an incrementing value which is initialized by the initialization vector, then XORing the output of the block cipher with the input plaintext. In the C implementation, the gctr_inc() function calls AES_encrypt() and increments the state of the counter as required by the AES-GCM standard.

3 GHASH Implementation
The GHASH portion of GCM essentially amounts to a multiplication by H in GF(2^{128}). Note that in AES, the Galois multiplication is by a constant, whereas in GCM the H multiplicant is derived from the key, so that both inputs to the multiplication operation can vary. Because of this, the implementation of the Galois multiplication in GHASH is much more complicated than in AES.

Originally, straightforward implementation of the Galois multiplication as described in [1] was used. However, this approach has an initiation interval equal to the number of bits in the multiplicands (128), which is significantly longer than the 10 cycles used by the AES encryption, making the Galois multiplication the limiting factor in throughput.

After attempting to solve this issue by applying HLS directives to pipeline the design and failing to achieve reasonable results, the C implementation of the Galois multiplication was revised. Based on [3], an optimized version of the Galois multiplication was developed in C which allows an initiation interval of 8 cycles under Vivado HLS (when synthesized in isolation).

4 Verification
In order to verify the C language implementation, a unit test program was developed in C which uses input vectors from the test vector files provided by [6] and presents these inputs to the circuit by repeatedly invoking the top-level aes_gcm() function. The outputs are then checked for correctness against the output test vectors.

C. Cryptol Implementation of AES
The Cryptol distribution comes with an implementation of AES in the Cryptol language, as well as a “test harness” of sorts that invokes the top-level AES function with a number of test vectors and verifies the results. However, the example implementation of AES is not directly synthesizable into HDL code, as it uses some language constructs that are not supported by the HDL backend. These portions of the example code were modified to use only supported language features and verify that the new implementation was still correct using the existing test harness, at which point it became possible to generate VHDL code based on the Cryptol description.

IV. SYNTHESIS OF HIGH-LEVEL DESIGNS
This section describes points of interest regarding high-level synthesis under each of the three frameworks used.

A Vivado HLS
This section describes the implementation of an FPGA design, based on the C implementation described previously.

1 Synthesis of VHDL implementation
Of the three HLS frameworks dealt with in this project, Vivado HLS seems to be the most full-featured and easy to work with. An IDE is provided that allows C or C++ code to be imported and edited. The process of translating that code into HDL can be controlled and constrained by the use of “directives,” either embedded in the code or specified interactively in the IDE. Doing the latter has the advantage of allowing different sets of directives to be applied to the same source code, with each set of directives comprising a separate “solution.” Thus, it is possible to create one solution that might attempt to optimize for performance, while another solution might optimize for minimal area.

Tweaking the directives can have a huge impact on the performance of the generated HDL code. By default, Vivado HLS leaves loops “rolled”, such that each iteration of each loop would consume at least one clock cycle. To execute multiple iterations of a loop in a single clock cycle (which is often desired in hardware), one must be sure to apply the “UNROLL” directive to the affected loops. Additionally, the “PIPELINE” directive can be used to automatically pipeline functions so as to achieve a particular initiation interval between subsequent inputs.

In order to achieve acceptable latencies, it was also necessary to apply the “INLINE” directive to most if not all functions that are called from the top-level aes_gcm() function. This was necessary because, like the top-level function, Vivado HLS treats all called functions as separate entities which have an interface that matches their C function signatures. This would mean, for instance, that even when calling a function that is entirely combinatorial, without inlining, additional latency cycles would be incurred due to handshaking between the caller and the callee. Notably, even when a function is not inlined, the hardware can only be shared between multiple call sites under particular circumstances (when the multiple call sites occur within the same function and at the same level in the call stack), so generally when optimizing for performance inlining is preferred.

Also, the initial C code was restructured such that when processing plaintext or ciphertext, the GHASH function for the previous block is computed concurrently with the AES-
128 encryption for the current block. This “compute GHASH one cycle later” modification should allow the two to execute in parallel and produce a latency related only to the maximum of the two, rather than the sum of the two, as would be the case when these operations are computed sequentially within the same function invocation.

Vivado HLS also includes support for “arbitrary-precision” integers in C and C++ code, allowing integer types of unusual sizes (for instance, “uint19” would represent an unsigned integer with 19 bits). While convenient for FPGA development, using these types requires Vivado’s “apcc” (arbitrary-precision C compiler) package to be used, limiting portability. Conversely, implementing unusually-sized integers with plain C code is possible (and portable), but tends to be tedious and error-prone, as simple operations like copying values or XOR-ing two values together must be coded as loops (which need to have the UNROLL directive applied in order to be implemented efficiently).

For this project, 128-bit values are used pervasively throughout the codebase, and I’ve used a mixture of arbitrary-precision and “normal” C types throughout the code. Initially, attempted to write the code entirely without arbitrary-precision types so as to be portable, but later began to use the “AP” types for convenience. Unfortunately, this meant that later when porting to LegUp, sections of the code that used AP types had to be rewritten.

2 Verification of VHDL implementation

For verification of the generated VHDL code, Vivado HLS has the ability to “co-simulate” hardware and software, with the hardware being driven by signals generated by a process executing in software. The C-based test program described earlier was used as the basis for a hardware/software co-simulation test in order to verify the correctness of the generated VHDL code. A pure-VHDL testbench was also developed which stimulates the HLS-generated circuit and verifies correct output.

B Cryptol

Cryptol is a domain-specific language designed for developing executable descriptions of cryptographic algorithms. Cryptol follows the “functional” language paradigm. The intent behind the Cryptol language is to give cryptographers a means to concisely communicate the details of a cryptographic algorithm in a form that is as straightforward as mathematical notation yet precise enough to be executed by a computer. Coincidentally, it has the interesting feature of being able to translate an algorithm described in Cryptol into HDL (Verilog or VHDL) code.

Cryptol can be executed in an read-evaluate-print interpreter mode, where Cryptol expressions can be executed interactively or code can be read from files. The read-evaluate-print mode was useful when becoming familiar with some of the language concepts. After modifying the example AES implementation, was able to generate corresponding VHDL code.

Notably, in 2014 an open-source version of Cryptol (dubbed Cryptol 2.0) was released. The Cryptol 2.x series of software appears to be a major, if not full rewrite of the previous, closed-source, Cryptol 1.x series. There are also some differences in the language itself between 1.x and 2.x. As of this writing, the 2.x series does not support translation to HDL, though this is in development. was able to obtain an academic license to Cryptol 1.8.27 for the purposes of this project.

completed FPGA implementation of the VHDL code generated by Cryptol using Xilinx ISE/XST.

C LegUp

The LegUp implementation was cloned from the Vivado HLS implementation and modified to remove use of the Vivado arbitrary-precision types so as to compile under LegUp. I’ve successfully been able to generate Verilog code using LegUp, but have not as of this writing yet been able to verify the correctness or viability of the generated code.

V. RESULTS

A AES-GCM Implementations in C

Using Vivado HLS, several different architectures were synthesized from the C AES-GCM implementation using different synthesis parameters (synthesis clock rate and directives). The results described in this section are based on post-place-and-route implementation in Xilinx ISE 14.6. The two implementations discussed in this section, “solution 3” and “solution 4”, are representative of the class of implementations achieved. Both of these include encompass all of the optimization and tuning steps discussed previously, and differ only in the following way:

Solution 3: HLS Synthesis clock period set to 10 ns.
Solution 4: HLS Synthesis clock period set to 100 ns.

1 Discussion of Solution 3

<table>
<thead>
<tr>
<th>FPGA Part</th>
<th>xcv6slx75-3csbg484</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>11,679</td>
</tr>
<tr>
<td>FFs</td>
<td>11,981</td>
</tr>
<tr>
<td>BRAMs</td>
<td>0</td>
</tr>
<tr>
<td>SLICEs</td>
<td>4,019</td>
</tr>
<tr>
<td>Clk Freq. (MHz)</td>
<td>125.0</td>
</tr>
<tr>
<td>Throughput (Mbit/sec)</td>
<td>50% PT/CT, 50% AAD: 359.6</td>
</tr>
<tr>
<td></td>
<td>50% AAD Only:</td>
</tr>
<tr>
<td></td>
<td>470.6</td>
</tr>
<tr>
<td>TP / Area (50/50 case)</td>
<td>0.0895</td>
</tr>
</tbody>
</table>

Table 1: Solution 3 Implementation Metrics

Table 1 shows the metrics characterizing the hardware realization achieved for solution 3. Note that the FPGA resource utilization is quite high, and would not even fit some smaller models in the Spartan 6 family.
As described in section IV, a functional testbench was developed to verify the correctness of the generated implementation. Since the top-level aes_gcm() function implements different operations on each invocation depending on its internal state, the timing waveforms generated by the testbench were used to determine the number of clock cycles required for each state of the progression, which are shown in Table 2.

<table>
<thead>
<tr>
<th>Initial State</th>
<th># cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction/Header:</td>
<td></td>
</tr>
<tr>
<td>IDLE_NOKEY, IDLE_KEYED, _WAIT_HDR, IV_WAIT</td>
<td>1</td>
</tr>
<tr>
<td>KEY_WAIT</td>
<td>56</td>
</tr>
<tr>
<td>AAD_WAIT</td>
<td>34</td>
</tr>
<tr>
<td>MSG_WAIT</td>
<td>23 for first block</td>
</tr>
<tr>
<td></td>
<td>55 for subsequent blocks</td>
</tr>
<tr>
<td>COMPLETING</td>
<td>87</td>
</tr>
<tr>
<td>COMPARE</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Solution 3 Latencies

These latencies are indicative of the underlying implementation. Remember that AES-128 encryption (involved in the GCTR operation) and Galois Multiplication (involved in the GHASH operation) form the primary computational components of AES-GCM. Note that an AES-128 encryption is involved in each of KEY_WAIT, MSG_WAIT, and COMPLETING, and that KEY_WAIT and AAD_WAIT involve a GHASH operation (and COMPLETING involves two GHASH operations). Remembering that the first MSG_WAIT state includes only a GCTR, we note that the number of cycles required for later MSG_WAIT states is roughly equal the number of cycles required for the first MSG_WAIT state plus the number of cycles for the AAD_WAIT state (which includes a GHASH). That is, $23 + 34 \approx 55$.

Based on this, we can see that in spite of the “compute GHASH one block later” optimization described previously, the code generated by Vivado HLS seems to fail to compute the current GCTR and previous GHASH in parallel, as we’d hoped. If this were the case, we’d expect MSG_WAIT for subsequent blocks to consume roughly 34 cycles (the max of GCTR and GHASH), but instead we see that it consumes close to the sum of both operations.

In Table 3, the number of clock cycles required to encrypt/decrypt an entire message of length $n$ blocks of plaintext/ciphertext and length $m$ blocks of AAD using solution 3 is given. Note that:

$$m = \lceil \text{bits}_{\text{AAD}}/128 \rceil, \quad n = \lceil \text{bits}_{\text{PT/CT}}/128 \rceil$$

Also, in the expressions below, the quantity $(n > 0)$ is defined as:

$$(n > 0) = \begin{cases} 
1 & : n > 0 \\
0 & : n = 0
\end{cases}$$

<table>
<thead>
<tr>
<th>Func.</th>
<th># of cycles $cylces(m, n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>$2 + 56 + 2 + 34m + (n &gt; 0) \cdot (23 + 55 \cdot (n - 1)) + 87$</td>
</tr>
<tr>
<td>AD</td>
<td>$2 + 56 + 2 + 34m + (n &gt; 0) \cdot (23 + 55 \cdot (n - 1)) + 87 + 3$</td>
</tr>
</tbody>
</table>

Table 3: Solution 3 Timing Formulas

Using the formulas above, the throughput is computed as:

$$\frac{\text{bits}}{\text{sec}} = \frac{f_{\text{clk}} \times 128}{\lim_{x \to \infty} \frac{cylces((1-p)x, px)}{x}}$$

where $p$ is the proportion of plaintext/ciphertext bits in the message, and $(1 - p)$ is the proportion of AAD bits. The $x$ variable which goes to infinity in the limit is representative of the sum of AAD and plaintext/ciphertext blocks.

In the case of solution 3, we find that this expression reduces to:

$$\frac{\text{bits}}{\text{sec}} = \frac{(125e6 \times 128)}{(34 \cdot (1 - p) + 55 \cdot p)}$$

which was evaluated at $p = 0.5, p = 1, \text{ and } p = 0$ to generate the throughput metrics in Table 1.

2 Discussion of Solution 4

As mentioned previously, solution 4 was generated by Vivado HLS based on the same codebase and directives, differing only in the HLS clock period. This caused Vivado HLS to generate a design that is not pipelined as deeply, significantly reducing the latency of the major parts of the computation, as can be seen in Table 5. Although the HLS clock period for solution 4 was 100 ns (10 MHz), by setting the implementation clock constraint to 10 ns (100 MHz), we were able to achieve an implementation with much better performance, summarized in Table 7.

<table>
<thead>
<tr>
<th>FPGA Part</th>
<th>xc6dx75-3csig484</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>11,562</td>
</tr>
<tr>
<td>FFs</td>
<td>5,284</td>
</tr>
<tr>
<td>BRAMs</td>
<td>0</td>
</tr>
<tr>
<td>SLICEs</td>
<td>4,596</td>
</tr>
<tr>
<td>Clk Freq. (MHz)</td>
<td>100.0</td>
</tr>
<tr>
<td>Throughput (Mbit/sec)</td>
<td></td>
</tr>
<tr>
<td>50% PT/CT, 50% AAD</td>
<td>1280.0</td>
</tr>
<tr>
<td>Plaintext/Ciphertext Only:</td>
<td>853.3</td>
</tr>
<tr>
<td>AAD Only:</td>
<td>2560.0</td>
</tr>
</tbody>
</table>

Table 4: Solution 4 Implementation Metrics

Notably, the “AAD only” throughput metric slightly outperforms the current Spartan 6-based AES-GCM implementation in GMU CERG's ATHENA database. The reason for this is clear when examining Table 5 below, where we see that the in the AAD_WAIT state (which computes the GHASH function using a Galois Multiplication) requires only 5 clock cycles.
high-level description in Cryptol. Implementation achieved using translation to VHDL from a particular target device is used in making this determination. Documentation claims that heuristic information about the pipelining depth determined? The Vivado HLS VHDL design for a particular clock period (for instance, how reasoning Vivado HLS uses when asked to synthesize a constraint is interesting, and leads one to question what implementation with a lower implementation clock period increasing the HLS synthesis clock period while running throughputs for solution 4.

Clock rate only drops by 20%, leading to a much better overall occurrence when moving from solution 3 to solution 4, but the decrease in pipeline depth, the individual terms and coefficients are smaller.

**Table 5: Solution 4 Latencies**

<table>
<thead>
<tr>
<th>Instruction/Header:</th>
<th># cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE_NOKEY, IDLE_KEYED, &quot;WAIT_HDR, IV_WAIT</td>
<td>1</td>
</tr>
<tr>
<td>KEY_WAIT</td>
<td>31</td>
</tr>
<tr>
<td>AAD_WAIT</td>
<td>5</td>
</tr>
<tr>
<td>MSG_WAIT</td>
<td>12 for first block 15 for subsequent blocks</td>
</tr>
<tr>
<td>COMPLETING</td>
<td>19</td>
</tr>
<tr>
<td>COMPARE</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 6: Solution 4 Timing Formulas**

Note that a significant decrease in the number of cycles occurs when moving from solution 3 to solution 4, but the clock rate only drops by 20%, leading to a much better overall throughput for solution 4.

The fact that a better implementation can be achieved by increasing the HLS synthesis clock period while running implementation with a lower implementation clock period constraint is interesting, and leads one to question what reasoning Vivado HLS uses when asked to synthesize a VHDL design for a particular clock period (for instance, how is the pipelining depth determined?). The Vivado HLS documentation claims that heuristic information about the particular target device is used in making this determination.

**B AES Implementation in Cryptol**

Table 7 shows the metrics achieved for the AES-128 implementation achieved using translation to VHDL from a high-level description in Cryptol.

**Table 7: Cryptol AES Performance Metrics**

<table>
<thead>
<tr>
<th>FPGA Part</th>
<th># cycles (per 128-bit encryption)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xc6vlx75t-3f784</td>
<td>20</td>
</tr>
<tr>
<td>LUTs</td>
<td>2093</td>
</tr>
<tr>
<td>FFs</td>
<td>1891</td>
</tr>
<tr>
<td>BRAMs</td>
<td>96</td>
</tr>
<tr>
<td>SLICEs</td>
<td>1485</td>
</tr>
<tr>
<td>Clk Freq. (MHz)</td>
<td>250.0</td>
</tr>
<tr>
<td>Throughput (Mbits/sec)</td>
<td>1600</td>
</tr>
<tr>
<td>TP / Area</td>
<td>1.08</td>
</tr>
</tbody>
</table>

**VI. CONCLUSIONS**

While the hardware implementations realized through this project are generally not competitive with existing implementations of AES-GCM (except for a slight advantage in the “AAD Only” for solution 4 as described above), there are still a useful few conclusions that can be drawn:

Of the three HLS tools investigated through this project, Vivado HLS seems well-positioned for use in both general-purpose computation and cryptography, which echoes the findings of [2]. Cryptol also appears well-suited for cryptographic hardware development, especially once the developing open-source “2.0” release begins to support a “VHDL backend.”

Vivado HLS is a powerful tool for developing hardware from a high-level description of an algorithm written in C, but the translation of such an algorithm into hardware is far from effortless. In addition to applying the appropriate synthesis directives, the appropriate care must be taken when structuring the C code so as to allow for efficient sharing of hardware. As was the case in this project, it may also be necessary to rewrite the high-level algorithm description so as to achieve efficient realizations in hardware. This is true both at the macroscopic level (as in the case of replacing the Galois multiplication function by the implementation based on Satoh, et. al.) and for the realization of even relatively simple, straightforward expressions.

Use of C language extensions for HLS coding such as arbitrary-precision integer types can be beneficial for enhancing the high-level code's clarity, but has significant portability drawbacks, even when switching between HLS frameworks.

The “HLS clock period” used by Vivado HLS when generating an HDL description from a top-level language implementation plays an important role in determining the architecture produced by HLS. We have found that in some cases Vivado HLS may be “too conservative,” adding unnecessary pipeline stages in the name of meeting the HLS clock period timing which in actuality only add latency. By tightening timing constraints on the later steps in the traditional implementation processes (Translate, Map, Place-and-Route), a significantly shorter clock period than the HLS clock period can be achieved with fewer pipeline stages, leading to an overall boost in throughput.

Using HLS in a “top-down” manner where HLS is immediately applied to the top-level function in a high-level language implementation may not be the best approach. Starting with lower-level “building block” functions and applying HLS to these functions in order to determine appropriate directives and parameters before composing the lower-level blocks to build the top-level functionality may allow the implementor an opportunity to recognize performance issues or inefficient translations earlier in the design process. Additionally, use of HLS to describe the entirety of a computational circuit, including both
interface/message parsing and the logic of the algorithm itself may not be the best approach to development. This again echoes the findings in [2], where it was found that one pervasive deficiency in the HLS-derived architectures was the inefficiency of the control unit.

ACKNOWLEDGMENT

The author would like to thank Ekawat Homsirikamol, who provided the HLS-ready C implementation of the AES cipher as well as a helpful example of Satoh’s sequential multiplier-adder for Galois multiplication. The author would also like to thank Adam Foltzer of Galois, Inc. for providing an academic license to Cryptol version 1.8.27.

1. REFERENCES

APPENDIX:

Simulation Timing Diagram of AES-GCM “Solution 3”
Simulation Timing Diagram of AES-GCM “Solution 4”