Crypto Accelerator

ECE 5775: High-Level Digital Design Automation

Andrew Dunne (asd222), Alex Katz (aik49), Daniel Wisner (daw268), Jacob Glueck (jng55)

December 11, 2018

1 Introduction

For this project, we implemented AES, RSA, SHA-512, and a Unix password cracking algorithm on an FPGA. RSA proved very difficult to implement, and ultimately was far slower on an FPGA than an optimized software version running on the ZedBoard's CPU – about 30 times slower for encryption. However, AES, SHA-512, and the Unix password cracker were very amenable to FPGA implementation. While neither was able to beat a server-grade Intel CPU (ecelinux) in terms of raw performance, all 3 achieved speedup with respect to the ZedBoard's CPU. AES had a speedup of about 1.5 times (depending on the variant of AES), while the password hasher had a speedup of 5.2 times. However, for AES and the password cracker, there are significant power advantages from using an FPGA. While the password cracker running on ecelinux processors required 0.0655 $\frac{\text{J}}{\text{Hash}}$, the optimized FPGA version required only 0.003 $\frac{\text{J}}{\text{Hash}}$.

For each algorithm, we implemented an optimized software version, and then an FPGA version. We benchmarked each algorithm on the server grade Intel CPUs (ecelinux), the embedded ZedBoard CPUs, and then on the ZedBoard FPGA. Each algorithm is explained in its own section below.

2 AES

The Advanced Encryption Standard (AES) is a widespread symmetric cryptosystem. AES is primarily used as a block cipher, which has openly exploitable parallelism by encrypting multiple blocks at once. Thus we saw AES as a prime candidate to optimize with hardware for large amounts of data. The AES implementation was based off of the NIST documentation[1] on AES as well as the github library `tiny-AES-c`[2].

2.1 Techniques

The AES algorithm consists of a number of rounds operating on a 4x4 byte matrix state to encrypt a 16-byte buffer input using a key. The key can be either 128, 192, or 256 bits wide. The larger the key, the more rounds are run on the state. Each round consists of four main operations called `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns`. Initially, the user-provided key is expanded a key schedule, called the `RoundKey`. During `AddRoundKey`, a portion of the `RoundKey` the state is updated by an XOR with a portion of the `RoundKey`. `SubBytes` is a one-to-one substitution of each byte value in the state using the Rijndael S-box. [3] `ShiftRows` is a rotation of rows in the matrix by differing offsets. Lastly, `MixColumns` operates on the columns of the state, performing some matrix multiplication with each column, but also additional Galois Field-related computations that are beyond the scope of this paper.

The algorithm described above encrypts a single input block. In order to exploit the high-level parallelism, we want to encrypt multiple input blocks at once. There are many standard encryption modes which specify how a multi-block message should be encrypted. Electronic Cookbook mode (ECB) encrypts each block individually, but is very insecure as identical blocks in the input message manifest as identical blocks in the output message. Cipher Block Chaining (CBC) XORs each block's plaintext with the encrypted prior block, forming a "chain". It is secure, but prevents parallel encryption, since each block depends on the prior block. Counter mode (CTR) solves this problem. It starts with a random *Initialization Vector* (IV, also known as a nonce). Block n is encrypted by XORing it with the encrypted value $IV + n$. This eliminates the dependencies between blocks,

[1] NIST guide to AES: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`.

[2] Embedded optimized tiny-AES-c library: `https://github.com/kokke/tiny-AES-c`.

[3] The S-box is constructed using finite field arithmetic, which is beyond the scope of this paper.

and still provides a secure cipher. This allows us to pipeline the encryption easily as each value being encrypted is one plus the previous value. Figure 1 shows with a diagram how AES CTR works, taking in 16-byte blocks and encrypting them with AES.

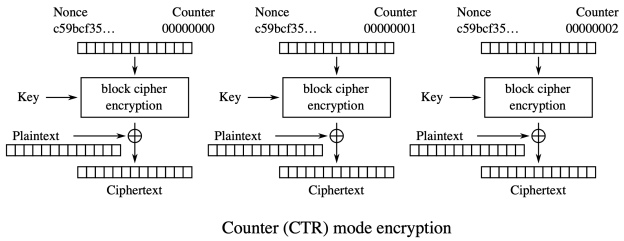

Figure 1: AES CTR diagram.

2.2 Implementation

Our optimized hardware synthesizable version was built using C++ and the Vivado `ap_int` library. Normally AES is implemented by taking a 16-byte array and converting that into a 4x4 byte matrix as the state, storing the bytes as in Figure 2. We packed the 16-byte block into one 128-bit number. The key is also stored as an X-bit number, where the X is determined by the key length. The RoundKey is packed into an array of 128-bit numbers, one for each of the 10 rounds. This makes the `AddRoundKey` function a single XOR with the current round's key schedule. The other functions (except `ShiftRows`), contain loops to loop over each byte or each column. These we completely unroll, as they can all be done in parallel, and it actually reduces area because indexing into the large `ap_uints` variably was more expensive than direct wires connected to the bits of the state. `ShiftRows` was already just a bunch of wire rearrangements and required no optimization. The four main functions had 0 or 1 cycle latencies, the 1 cycle for indexing into the Sbox or the RoundKey arrays. Ideally we could fully partition those, and reduce those latencies to just combinational logic, but this took too much area.

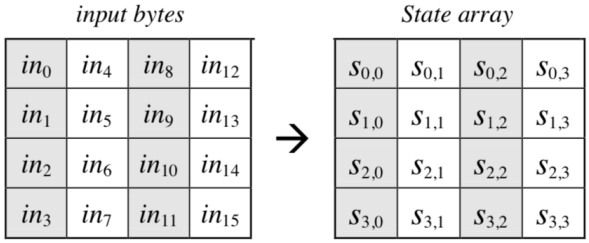

Figure 2: AES converting input into state.

In our top level, we have a loop that reads in values, ciphers the counter, XORs to get the plaintext and writes this value out. We pipeline this loop, as the Cipher function is inlined.

2.3 Evaluation

In order to evaluate our optimized FPGA version, we created a host program that reads 8000 bytes of data (500 blocks), sends the key, initiation vector, and data to the FPGA, and then reads back the encrypted data and compares that to the pre-saved encrypted data for testing. Based on our synthesis report, our pipelined block loop had an II of 4 and a depth of 46. 500 blocks is thus sufficient for measuring speedup as it well surpasses the warm up and cool down times of the pipeline. The final area usage was very low, reaching only about 37% of the LUTs. The full area usage for AES-256 is in Table 2.3. The final estimated latency from the synthesis report was 2089 cycles for AES256. Unfortunately, the limiting factor on the latency came from the interface, which could only read one 1 32 bit wide integer per cycle. This resulted in our II of 4, as each block required 128 bits (4 reads). Had we been able to read all 128 bits in one cycle, the design would have acheived an II of 1.

	BRAM_18K	DSP48E	FF	LUT
Utilization (%)	15	0	6	37

Table 1: The final area usage report of AES-256 (other key-lengths are very similar).

For a final comparison, we passed the same 8000 bytes through the `tiny-AES-c` embedded processor optimized version of AES on both the ecelinux servers as well as the ZedBoard's microprocessor. The results of this test are in Table 2.3. Both ecelinux runs beat the FPGA by an order of magnitude, which is largely expected. The FPGA does successfully beat the ZedBoard's ARM processor, with a resulting speedup of about 1.75x. The FPGA baseline times are estimated because they wouldn't actually synthesize due to too much area usage. As mentioned above, had we not used the xillybus interface, and been able to transfer 128 bits per cycle, the design would have been close to 500 cycles or 5ms per block. This time is the same order of magnitude as ecelinux, which would be extremely impressive, especially considering power.

Test System	Time (ms)			Speedup		
	AES128	AES192	AES256	AES128	AES192	AES256
ecelinux-sw	1.6	1.9	2.1	0.08	0.09	0.12
ecelinux-csim	5.1	6.2	7.2	0.26	0.30	0.41
zedboard-sw	25.5	30.0	34.3	1.32	1.46	1.96
fpga-baseline	4560.0	5820.0	4120.0	236.2	283.9	234.5
fpga-opt	19.3	20.5	17.5	1	1	1

Table 2: The results of the tests run for evaluation.

Overall, these results bode well for a hardware implementation of AES. In embedded scenarios where fast AES is needed for large amounts of data, the FPGA could be extremely useful. The Xilinx claims ZedBoard only uses sub-Watt power (we will assume 1 Watt) in comparison to ecelinux which is rated up to 120 Watts (we only make use of one core, thus 15 Watts/core). Looking at power per performance, where performance is the number of blocks per second, our ZedBoard achieves 0.035 J/block for AES256 and ecelinux uses 0.063 J/block. The ZedBoard implementation beats ecelinux in energy efficiency as ecelinux uses about 1.8 times the amount of energy. Intel's x86 ISA does include an AESENC instruction that performs one round of AES encryption. Using this instruction would most likely use comparable energy to the FPGA. Another thing to consider is as we send more data, the bandwith required to read and send this data will increase. In a situation where an embedded processor is used, and power is a major constraint, using the FPGA could be a viable option as it provides a speedup while keeping power usage down at the expense of more area.

3 RSA

RSA is a widely used public-key cryptosystem. It operates as a block cipher, encrypting a fixed length amount of data at once. Most RSA used today is 1024 - 4096 bits (which is the length it bits of the modulus, and also the approximately the number of bits that can be encrypted per block). For this project, we focused on 1024 bit RSA.

An RSA key pair consists of a public key (n, e) and a private key (n, d). n is the modulus, e is the encryption exponent, and d is the decryption exponent. n is generally a very large number; the number of bits in n defines the bitsize of the algorithm. For this project, n is a 1024 bit integer, composed of the product of two primes p and q. e and d satisfy the requirement that $e \cdot d = 1 \mod \varphi(n)$, where $\varphi(n)$ is the Euler Totient function.

Encryption of the plaintext message m is performed by modular exponentiation to produce the ciphertext c: $c = m^e \mod n$. Decryption is performed in the same manner: $m = c^d \mod n$. In order to maintain a 1 to 1 mapping between input messages and output messages, $m < n$. This sets the block size: if n is a 1024 bit number, then essentially 1023 bits can be encrypted at once.

For this project, we explored accelerating *modular exponentiation*, allowing the encryption and decryption of blocks to be computed on an FPGA.

3.1 Techniques

Implementing modular exponentiation required two main components: the actual computation, and algorithms to perform arithmetic on very large numbers.

The fundamental modular exponentiation algorithm is displayed in Algorithm 3.1.

Algorithm 1 Modular exponentiation algorithm.[4]

```
procedure MODULAR_POW
    if modulus = 1 then return 0
    result ← 1
    base ← base mod modulus
    while exponent > 0 do
        if exponent mod 2 = 1 then
            result ← (result · base) mod modulus
        exponent ← exponent >> 1
        base ← (base · base) mod modulus
    return result
```

This function is trivial to implement, except that it requires performing multiplication and modulus operations on very large numbers. In particular, since all the numbers involved are 1024 bits, it required performing 2048 bit arithmetic as the intermediate results could be over 1024 bits.

We represented the 2048 bit numbers 32 digit base 2^{64} numbers. [5] In this notation, a number a is represented as:

$$a = a_{31} \cdot \left(2^{64}\right)^{31} + \ldots + a_2 \cdot \left(2^{64}\right)^{2} + a_1 \cdot \left(2^{64}\right)^{1} + a_0 \cdot \left(2^{64}\right)^{0}$$

The coefficients $(a_{31}, \ldots, a_2, a_1, a_0)$ are the digits of a in base 2^{64}, and are stored in an array.

Using this representation, multiplication and division can be performed much in the same way they are performed on paper using "long multiplication" and "long division". For an in depth explanation of the algorithms involved, see the references above.

3.2 Implementation

The entire RSA system is implemented in C++, in the `rsa` directory. It uses a mix of both hardware and software, using hardware for modular exponentiation, and software for key generation, chunking, and padding. For the hardware portion, we targeted the ZedBoard (`xc7z020clg484-1`).

Figure 3 shows the division of work between hardware and software for an encryption flow. The "CBC" software operation refers to Cipher Block Chaining, which is used to turn RSA, which is a block cipher, into a secure stream cipher. Instead of encrypting each block directly, the XOR of each block with the encrypted previous block is encrypted. For the first block, the prior block is replaced by a random initialization vector. This ensures that duplicate blocks in the input do not result in duplicate blocks in the output.

On the FPGA, the top-level functions reads three 1024 bit numbers (the base b, the exponent e, and the modulus n) from an input stream, computes $b^e \mod n$, and then writes the result the output to a stream. This top-level module is defined by the `dut` function in `rsa/fpga_rsa.cc`. Communication between the host and FPGA on the ZedBoard is done using the `/dev/xillybus_read_32` and the `/dev/xillybus_write_32` character devices.

The software portion, `rsa/rsa.cc`, takes a message, performs all the key generation, chunking, and padding, and then encrypts each block using either the FPGA or software.

In addition to the FPGA implementation, we also had 2 software implementations: "opt" and "sim". The "opt" implementation used the GNU Multiple Precision Arithmetic Library (GMP) to perform modular exponenti-

[4] The "Right-to-left binary method" from Wikipedia: `https://en.wikipedia.org/wiki/Modular_exponentiation`.

[5] This representation is based directly off code from possibly-wrong: `https://github.com/possibly-wrong/precision/blob/master/math_Unsigned.h`. The code was heavily modified to make it amenable to HLS. *Mathematics and Algorithms for Computer Algebra* by Francis J. Wright was instrumental in making the required modifications (`https://people.eecs.berkeley.edu/~fateman/282/F%20Wright%20notes/week4.pdf`). The original implementation off all these algorithms comes from Donald Knuth's *Art of Computer Programming, Volume 2: Seminumerical Algorithms.*

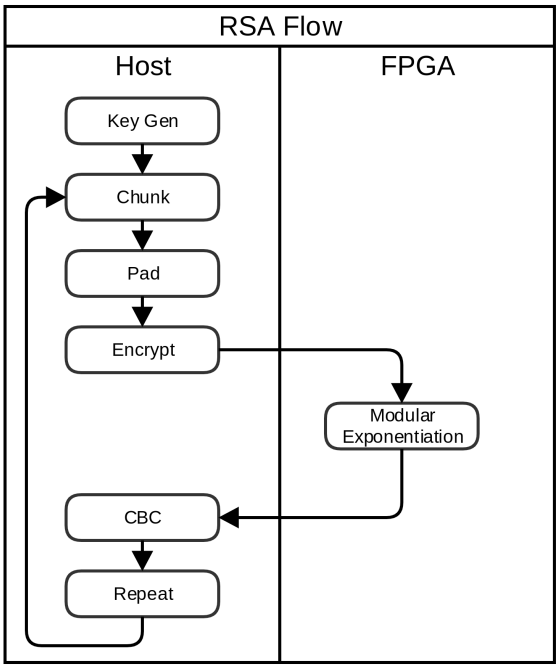

Figure 3: RSA hardware-software system

ation. This library is highly optimized. The "sim" implementation ran the code to be synthesized directly as C++ code, simulating exactly the computations carried out on the FPGA.

One unique aspect of our system is that switching between the different implementations of modular exponentiation (either optimal software, simulated software, or using a real FPGA) can be done with a simple macro definition while compiling `rsa.cc`. When using the provided Makefile (`rsa/Makefile`), the desired mode can be set as follows:

- Optimal software: `make`. Produces the executable `rsa`.
- Simulated software: `make MODE=FPGA_SIM`. Produces the executable `rsa-sim`.
- Real FPGA: `make MODE=FPGA_REAL`. Produces the executable `rsa-host`.

3.3 Evaluation

We evaluated both software versions ("opt" and "sim") on both the ZedBoard and ecelinux machines. The FPGA design was evaluated on the ZedBoard. Due to the complexity of the design, there is not a significant baseline design to compare it with. Without the optimization directives, it was difficult to even get a design that met timing closure. Table 3.3 shows the timing for each implementation on each computer. The speedup columns (or rather, slowdown) show the speedup relative to the "fpga-opt" implementation. The "cosim" row indicates the expected time as given by the HLS co-simulation tool.

Overall, the software implementations are far better than the FPGA implementation. The optimal software implementation running on ecelinux ("ecelinux-sw-opt") was very fast, about 380x faster than the FPGA implementation for decryption. This is because the GMP library is very well optimized. Essentially, it is not even using the same algorithms we accelerated, but instead running far more advanced and complicated algorithms. Even on the ZedBoard, which has a much slower processor than ecelinux, the optimized software algorithm is about 30 times faster for encryption than the FPGA. The simulated software algorithm (which is running the HLS code) is faster, at least for decryption, than the ZedBoard CPU.

Overall, however, RSA was really difficult to accelerate. Large number arithmetic has very little obvious parallelism, and is fraught with dependencies. Furthermore, there is no reason to accelerate it. Since it is

known to be slow, most modern cryptographic algorithms use RSA only to exchange a key for a symmetric algorithm such as AES, and then use AES for the bulk of the data transfer. PGP is one such algorithm. Given that essentially one RSA block is only used for a message, and the bulk is another algorithm, it is wasteful to specialize FPGA area for RSA.

Version	Encrypt (ms)	Decrypt (ms)	Speedup Enc	Speedup Dec
ecelinux-sw-opt	0.0355	0.627	0.00306	0.00261
ecelinux-sw-sim	1.6421	112.7	0.14161	0.46966
zedboard-sw-opt	0.3997	19.478	0.03447	0.08117
zedboard-sw-sim	8.1581	576.2225	0.70353	2.40133
fpga-opt	11.596	239.96	1	1
cosim	3.918	234.23	0.33866	0.97612

Table 3: RSA algorithm runtimes

4 SHA512

The Secure Hash Algorithm (SHA) is a cryptographic hash function used in a wide range of applications, from checking file integrity to securely storing passwords. At a high level, SHA is a function from an arbitrary size message into a fixed-size hash. Furthermore, the function is very difficult to invert: given a hash, determining the original message is nearly impossible. The original SHA algorithm, SHA-1, produced 160-bit hashes, while SHA-512 produces 512-bit hashes.

We implemented SHA-512 on an FPGA, used it to accelerate Unix password hashing, and then implemented a simple brute-force hardware accelerated Unix password cracker.

4.1 Techniques

4.1.1 SHA-512

The core of SHA-512 is the compression function, which takes an intermediate hash value and a single 1024-bit block of data and returns another hash value. For the first block, the hash function is called with a fixed initial value. Then for each subsequent block, the intermediate hash passed into the compression function is the result of the previous run of the compression computation. That is, any given intermediate hash value directly depends on the preceding intermediate hash.

At the end of the message, the algorithm appends a single `0x80` byte. The last 128 bits of the final block are set to the length of the message in bits, as a 128 bit big-endian number. The space between the `0x80` and the start of the length is zero-padded. The final hash is just the hash computed for the last block.

The algorithm for the compression function is given in 4.1.1. Let H^i denote the current intermediate hash value, and H^i_n be the n^{th} 64-bit word in that hash. Let M^i denote the i^{th} 1024-bit chunk of the message, and let M^i_j denote the j^{th} 64-bit word in that chunk (treating it as big-endian). For our purposes, the exact definitions of $\Sigma_0, \Sigma_1, \sigma_0, \sigma_1, Maj(a,b,c), Ch(e,f,g)$ are not relevant. It is sufficient to understand that they are various bit twiddling functions involving many shift and XOR operations.

4.1.2 Unix Password Hashing

Most unix systems store the passwords hashes in the `/etc/shadow` file. Entries in this file have the format: `[user]:$[hash_algorithm]$salt$hash`. Modern systems usually use SHA-512 for the underlying `hash_algorithm` which is represented with a `6`. The algorithm for hashing a Unix password is shown in 4.1.2. This algorithm is normally accessed through the `crypt()` function in `<unistd.h>`.

Note that the SHA-512 Unix password hashing algorithm requires computing 5000 SHA-512 hashes of variable length messages, where each hash is dependent on the previous one. It is extremely slow operation; the ZedBoard's ARM core can only hash about 10 passwords per second.

Algorithm 2 SHA-512 Procedure for hashing a block. Adapted from `http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf`

procedure HASHBLOCK
- $a \leftarrow H_1^{i-1}$ ▷ Assign $a..h$ to the corresponding 64-bit chunks of the intermediate hash
- $\vdots$
- $h \leftarrow H_8^{i-1}$
- $W_j \leftarrow M_j^i$ for $j = 0, 1, ..., 15$ ▷ The first 16 64-bit words of W are from the message block
- **for** $j = 16$ to 79 **do** ▷ The remaining 64 are computed recursively
  - $W_i \leftarrow \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{j-15}) + W_{j-16}$
- **for** $j = 1$ to 79 **do** ▷ Apply the compression function to compute the output hash
  - $T_1 \leftarrow h + \Sigma_1(e) + Ch(e, f, g) + K_j + W_j$
  - $T_2 \leftarrow \Sigma_0(a) + Maj(a, b, c)$
  - $h \leftarrow g$
  - $g \leftarrow f$
  - $f \leftarrow e$
  - $e \leftarrow d + T_1$
  - $d \leftarrow c$
  - $c \leftarrow b$
  - $b \leftarrow a$
  - $a \leftarrow T_1 + T_2$
- $H_1^i \leftarrow a + H_1^{i-1}$ ▷ Assemble the new hash using the computed values
- $\vdots$
- $H_8^i \leftarrow a + H_8^{i-1}$

Algorithm 3 Unix SHA-512 *crypt*() implementation. Adapted from `https://akkadia.org/drepper/SHA-crypt.txt`

procedure CRYPT_SHA512()
- B = sha512(password + salt + password)
- A = sha512(password + salt + B[:pwlen])
- DP = sha512(password * pwlen)
- DS = sha512(salt * (16 + A[0]))
- **for** $j = 1$ to 4999 **do**
  - buf = []
  - buf.append(DP[:pwlen] if (i % 2) else A)
  - buf.append(DS[:saltlen] if (i % 3) else)
  - buf.append(DP[:pwlen] if (i % 7) else)
  - buf.append(DP[:pwlen] if (i % 2 == 0) else A)
  - A = sha512(buf)
- return base64Encode(A)

4.2 Implementation

4.2.1 SHA-512

We implemented SHA-512 in software first, using the algorithm presented in the "Descriptions of SHA-256, SHA-384, and SHA-512" (see algorithm 4.1.1) We then modified it as needed to make it synthesizable. Ultimately, we ended up with *SHA512Hasher*, a C++ class that exposes the following methods:

```
void reset(); // Reset the state of the hasher
void update(const uint8_t *msgp, uint8_t len); // Append msg to the hasher
void byte_digest(uint8_t buf[64]); // Output the SHA-512 hash
```

Internally, the class contains a length 128 byte buffer (to store the current block) as well the current intermediate

hash, which is initialized to the initial SHA-512 intermediate value. This buffer is filled as `update()` is repeatedly called on it. Whenever the buffer fills up, the class automatically (in the `update()` call) runs the SHA-512 algorithm for the buffered block and computes the next intermediate hash value based on this current block and the stored intermediate hash hash of previous block. This was implemented as a private method called `hashBlock()`. When `byte_digest()` is called, the hasher does the final steps for the last block, and computes the final SHA-512 hash. A `reset()` method was also provided to allow the same hasher instance to be reused to compute further hashes.

This implementation works extremely well for an HLS FPGA implementation because the entire message does not have to be stored. Only a temporary 128 byte buffer to store the current block, and a 64-byte buffer to store the intermediate hash for the previous block is needed. Furthermore, by doing this, it allows the size of the message to be variable and be unknown at compile time.

Since the slow part of the algorithm was was performing the 80 SHA-512 rounds, optimizing this `hashBlock()` method was the first priority. Initially, its latency was 534 cycles. However, the optimized version was only 88 cycles.

One of the most important optimizations involved recognizing that the i^{th} value of W (algorithm 4.1.1) only relies on the 15 preceding W values. Since each block of W is accessed sequentially, we can use a 16-block block array as a shift register to store the values we need, instead of the entire 80 block array. Since the first 16 values in W are just the current message chunk with no additional processing, we can simultaneously do the first 16 rounds of the compression function and also fill W in parallel, and then do the remaining 64 rounds computing one new W value each time. We fully partitioned this W array, and removed the need to read/write from a BRAM each cycle. Furthermore, by implementing W with a shift register, each loop iteration would be reading and writing to the same registers every time, as opposed to a different BRAM address.

The final set of optimizations involved partitioning the 128 byte block buffer cyclically by a factor of 8. This allowed the first length 16 loop discussed in the previous paragraph to initialize each W_i in a single cycle. Next, the intermediate value hash array was partially partitioned so that all $a \ldots h$ could be initialized at the beginning of `hashBlock()` as well as updated at the end in a single cycle. Finally, both the 16 and 64 iteration loop were pipelined, and due to the previous optimizations, allowed them to have an II of 1.

At this point `hashBlock()` had a latency of 88 cycles, which was quite good considering the SHA-512 algorithm itself consists of 80 iterations, where each iteration is dependent on the previous one. Thus, after these optimizations, there was practically no further parallelism that could be exploited.

4.2.2 Unix Password Hashing

The entirety of the SHA-512 portion of the `crypt()` function was implemented on the FPGA in `unix_cracker.cpp`. The implementation was relatively straight forward, and closely resembles the pseudo code in algorithm 4.1.2. Due to the lack of remaining resources on the Zynq's FPGA, only minor optimizations were applied to this portion of the implementation. Additionally, the vast majority of the compute time was spent in the `SHA512Hasher` methods, so resources were conserved and instead used to enable the `SHA512Hasher` optimizations explained previously.

A basic host program was written, `zedboard/host.cpp`, which sends a salt and password (null-terminator separated) over the Xillybus. The FPGA then sends the resulting hash back. The host program is responsible for generating password guesses, sending to the FPGA, then comparing the returned result against the known hash extracted from the `etc/shadow` file. If these two hashes match, then the password tested was correct.

4.3 Evaluation

In order to evaluate our password hasher, we tested it on three different platforms: the Intel CPU on an ecelinux server, the ARM CPU on a ZedBoard, and the ZedBoard FPGA. For the evaluation we used a host program that simulated using the password hasher for a dictionary attack. The host program reads passwords from a text file of commonly used passwords, and then sends the password as plaintext to the FPGA, which then sends back the hash of the password as a base64 number represented with ASCII characters. For evaluating the performance of the program on the ecelinux and ZedBoard CPUs, we instead used the `crypt()` from `<unistd.h>` function provided by the operating system to generate the hashes. On ecelinux, we found that our test program was

Version	Latency (cycles)	BRAM Usage	DSP Usage	FF Usage	LUT Usage
baseline	534	2%	0%	0%	5%
baseline-pipeline-1	299	2%	0%	1%	6%
baseline-pipeline-2	235	<1%	0%	9%	20%
shift-register	2373	2%	0%	1%	9%
shift-register-pipeline-1	91	<1%	0%	3%	12%
final-opt	88	<1%	0%	4%	7%

Table 4: Usage and cycle counts for different version of our SHA-512 implementation. The baseline versions are nearly identical to the software implementations, but they make use of optimization directives. The shift-register versions use the optimization described in the Techniques section to reduce the size of W and to remove the extraneous loops, in addition to making use of optimization directives.

Version	Average Time (s)	Hash/sec	Speedup	Price/Perf. ($/Hash/sec)	Power/Perf. (J/Hash)
ecelinux	0.00346	229	24	7.64	0.0655
fpga-opt2	0.0200	50.1	3.5	5.99	0.00200
fpga-opt1	0.0300	33.3	5.2	9.90	0.00300
zedboard	0.104	9.62	1	31.2	0.00515

Table 5: Results for the Unix Password Hasher. The ecelinux and ZedBoard versions were identical software implementations, compiled with -O3. The fpga-opt1 and fpga-opt2 were differently optimized FPGA implementations of the password hashing algorithm. The speedups are given relative to the ZedBoard ARM CPU. The Price/Performance column is the ratio of the cost of the CPU to the number of hashes they computed per second. Prices are from manufacturer's recommended pricing. The Intel CPU cost $1750 while the entire ZedBoard (ARM CPU and FPGA) cost $300. The Power/Performance column is the ratio of the average power consumption to the number of hashes they computed per second. The average power consumption information was taken from the manufacturers' websites. The Intel CPU has a TDP of 120W, and Xilinx claims the ARM CPU and FPGA both have sub-watt TDPs of approximately 100mW. We divided the TDP by the number of cores for the Intel and ARM CPUs, and are assuming a linear speedup with multiple cores.

able to hash 230 passwords per second. While this may seem low, remember that each call to `crypt()` is computationally expensive since it performs 5000 rounds of SHA-512. On the ZedBoard's ARM CPU, we could only perform 10 hashes per second. The difference in speed is surprising, given that the ZedBoard CPU only operates at half the clock speed of the ecelinux CPU, and that `crypt()` is not parallelizable, so it doesn't benefit from the larger number of cores available on ecelinux. Most likely it is due to differences in the implementation, the x86 version might use special instructions that are not in the ARM ISA to speed up the computation. The optimized version of the FPGA, described in the previous section, was able to achieve 50 hashes per second, which is over a 5x improvement over the ZedBoard CPU. Even though the Intel CPU beats the FPGA in terms of raw power, it loses when it comes to the ratio of price/performance and power/performance. The most optimized version of the FPGA hasher achieves a price-to-performance ratio of 5.99, while the Intel CPU's ratio is 7.64 (lower is better). The most striking difference is in the power-to-performance scores. The Intel CPU has a ratio of 0.524, while the FPGA achieves .00200, *nearly two orders of magnitude lower* than the Intel CPU. This our makes the FPGA-based implementation the clear choice for embedded applications, where power consumption is a major concern.

5 Project Management

The initial version of the project called for us to implement 4 algorithms: Elliptic Curve Cryptography, RSA, AES, and SHA. Alex was supposed to do elliptic curve, Jacob RSA, Drew AES, and Aaron SHA. However, elliptic curve cryptography proved to be difficult to implement, so Alex joined forces with Aaron to work on a Linux password cracker. From that point on, we all worked individually on our own projects and got them working. The following list details the work performed by each member:

- **Drew** (AES)
  - Implement simple software AES (2018-11-07).
  - Synthesizable baseline AES (2018-11-15).
  - Optimizing AES by using apuints and rewriting (2018-11-20).
  - Optimized CTR AES complete (2018-11-29).
- **Jacob** (RSA)
  - Implement optimized software baseline RSA using GMP (2018-11-07).
  - Implemented very bad RSA baseline (2018-11-15).
  - Removed most variable bound loops from algorithm (2018-11-20).
  - Work on meeting timing closure and optimized version (2018-11-29).
- **Aaron** (SHA)
  - Optimized SHA256 for CPU (2018-11-07).
  - Synthesizable SHA256 (2018-11-07).
  - Synthesizable SHA512 (2018-11-20).
  - Created Unix password cracker (5000 rounds with special salting) (2018-11-29).
- **Alex** (SHA)
  - Investigate Elliptic Curves, decide on Secp256k1 curve (2018-11-07).
  - Abort ECC, switch to password cracker. Design simple brute force algorithm (2018-11-15).
  - Explore F1 (2018-11-29).
  - Write host program for password cracker (2018-11-29).

6 Conclusion

In this paper we implemented three separate FPGA-accelerated cryptography projects The first algorithm, AES, a symmetric key cryptography algorithm, was a good candidate for FPGA acceleration, since you can encrypt any number of blocks in parallel. In practice, we saw that our FPGA-accelerated implementation beat the ZedBoard ARM CPU outright. While it did not beat the ecelinux Intel CPU in raw performance, it did outperform it in price-to-performance and power-to-performance ratios. Our second algorithm, RSA, is a public-key cryptosystem. It was not amenable to an FPGA implementation, as there was little parallelism to exploit. It was far slower than an optimized software version, even on the ZedBoard. However, RSA is also not a good candidate for acceleration because it is most often used not to encrypt large amounts of data, but to encrypt keys to allow a symmetric algorithm, like AES, to encrypt large amounts of data. Thus, the potential speedup gained by accelerating RSA is fairly small. Our last project was to implement a Unix Password Cracker based around SHA-512. The process for hashing Linux passwords is computationally expensive and centers on 5000 rounds of SHA-512. We offloaded this entire computation onto the FPGA, and then used a host program to send commonly used passwords to the FPGA to hash in a dictionary-style attack. Our implementation beat the ZedBoard ARM CPU and lost to the ecelinux Intel CPU, but like AES, our FPGA-based design is significantly better than the other platforms tested in terms of price-to-performance and power-to-performance.