An Automated Scheduler-based Approach for the Development of Cryptoprocessors for Pairing-Based Cryptosystems

Theodore Winograd, Rabia Shahid, and Kris Gaj
Department of Electrical and Computer Engineering
George Mason University
Fairfax, VA, USA
twinogra@gmu.edu, rabia.shahid4@gmail.com, kgaj@gmu.edu

***Abstract*—Pairings have been used to develop a wide range of cryptographic protocols to solve problems beyond the reach of traditional public-key cryptographic schemes. At the same time, they are mathematically complex and involve a rich hierarchy of operations over finite fields and extension fields. Performing a pairing operation involves large number of modular multiplications, additions, and subtractions that require cumbersome manual scheduling of operations. Due to large sequence of operations and custom hardware architectures designed for a specific pairing algorithm, it becomes infeasible to change the underlying pairing type, elliptic curves, or even parameter sets without redesigning the entire architecture. We present the design of a configurable and generic execution unit, capable of being used for any pairing algorithm, and serving as a coprocessor to perform all the underlying operations. The execution unit is supported by a software static scheduler to automate the process of manual scheduling of operations at the lowest level of hierarchy, i.e., at the level of prime field arithmetic. We develop a hierarchical input format to support the tower of fields used to represent extension field elements involved in pairings. We optimize the overall performance of the cryptoprocessor by using an optimal number of multiplier units, capable of taking full advantage of the parallelism present in the algorithm and a single modular adder/subtractor, working in parallel with multipliers.**

I. Introduction

Pairing-based Cryptography (PBC) [1], [2], [3], [4] was first shown to have practical use for identity based encryption (IBE) by Dan Boneh and Mathew K. Franklin in 2001. In the majority of pairing-based schemes, the most vital operation is the computation of bilinear maps: a pair of elliptic curve points is mapped to a multiplicative group of a finite field, serving as the primary operation for the cryptosystem. Faster pairing-based cryptoprocessors are important as they ensure pairing-based schemes can be used in practical communication protocols. these cryptoprocessor can take advantage of algorithmic improvements devised by mathematicians and cryptographers, which can aim to reduce the time of basic finite field operations. Researchers have also been working on finding pairing algorithms and elliptic curve forms that reduce the overall computational complexity. Many optimizations in preliminary pairing algorithms have been proposed to make them suitable for implementations on a given platform. The improvements, such as reduced loop lengths, the use of distortion maps, twisted curves, tower field extensions, and different coordinate systems, are all targeted towards making pairing implementations feasible.

Cryptographic engineers are also working to make efficient implementations of these algorithms on FPGAs and ASICs. Unfortunately, the researchers working on performance and security improvements are often mathematicians, cryptographers, or hardware engineers, specializing in their respective fields-limiting the ease at which breakthroughs can be distributed between groups. The theory behind pairing algorithms, going beyond the concepts involved already in Elliptic Curve Cryptography, is challenging for most hardware engineers. Therefore, a significant gap exists between cryptographers/-mathematicians, who design these algorithms, and engineers, who implement them.

We propose a new methodology and tools to help bridge this gap. Based on the platform and application, engineers can use our framework to choose which pairing algorithm is the best for the respective requirements and estimate its performance in terms of timing and resource utilization. As described in [5], there are instances when hardware designers have to treat pairings as black boxes while implementing cryptographic schemes based on pairings. In these scenarios, it is easy to make invalid assumptions due to insufficient background knowledge, which can result in developing cryptographic schemes that cannot be realized in practice and may have weak security properties. More generally, our cryptoprocessor is paired with a scheduler that generates a sequence of instructions based on the underlying field arithmetic–and can adjust to any other cryptosystem based on finite field arithmetic. The overall design is implemented taking into consideration the trade-off between the minimum resource utilization and low computational time, combined with configurability of arithmetic units.

II. Related Work

After the development of the first practical IBE scheme [1], attention was driven to implementations of pairing algorithms on different platforms. These efforts led to the emergence of new pairing algorithms (Eta, ate, X-ate, R-ate, optimal pairings), which could be applied to different elliptic curves as in [6], [7], [8] (supersingular curves, MNT curves, Barreto

TABLE I: Algorithmic choices affecting implementation efficiency of Pairing Based Cryptosystems.

Property/Parameter	**Choices**
Security Level	80, 128, 192, 256
Type of Pairing	Type 1, Type 2, Type 3, Type 4
Pairing Algorithm	Eta, Tate, ate, optimal ate twisted ate, X-ate, R-ate
Finite Field	Prime
Class of Elliptic Curve	Supersingular, Ordinary
Embedding Degree	$2 \leq k \leq 60$
Elliptic Curve Form	Weierstrass, short Weierstrass, Barreto-Naehrig (BN), Edwards, Twisted Edwards, Hessian, Huff, Generalized Huff, Jacobi Intersection, Jacobi Quartic, Montgomery, Selmer
Degree of Twist	2, 4, 6
Special Primes	Pseudo-Mersenne, Generalized Mersenne, Solinas

Naehrig (BN) curves, Edwards curves, etc.). It also led to the use of binary, ternary and prime fields to construct pairing algorithms. As pairings are considered to be computationally expensive, implementations of pairing algorithms on different platforms were necessary to analyze and determine their feasibility for practical applications.

With the finding that binary fields are no longer considered secure, the use of optimal pairings has assisted in achieving faster implementations over prime fields. To date, the number of hardware implementations of pairings over prime fields is considerably smaller than the number of corresponding software implementations-yet the hardware implementations provide significant performance benefits over their software counterparts.

These algorithms can be quite hard to implement without extensive knowledge required to understand the operations at different layers. The choice of parameters that makes things affects the specific order of operations for the implementation. Hardware implementations that rely on manual scheduling of operations based on optimal ate pairings are reported in [9], [10], [11], [12], [13]. The fastest to-date implementation of a pairing algorithm over prime fields was reported by Yao et al. in [9]. Their pairing coprocessor uses Residue Number System (RNS), lazy reduction generalized in [11], Karatsuba technique to reduce the number of multiplications in the extension fields, and pipeline scheduling. They improve their own previous implementation, reported in [11], by reducing the number of moduli in the RNS basis, enabling faster reduction than before. Their design is specific to optimal ate pairing, and they perform manual scheduling to explore parallelism in RNS arithmetic, which is a very cumbersome task.

To the best of our knowledge, all hardware designs reported in the literature on implementing pairing algorithms are manually scheduled to explore parallelism, and are therefore specific to a single pairing algorithm. We developed a framework that allows the user to generate a scheduled sequence of operations for new curve forms. The user can also estimate the performance by configuring different number of multiplier cores to find the sweet spot between the maximum performance and minimum resource utilization. This approach will allow the pairing based cryptographic community to determine if any of the newer algorithmic models are suitable for implementations on a given platform by allowing quick prototyping.

Performing efficient modular arithmetic is also crucial for the overall performance in such applications. Suzuki in [14], [15] combined the Multiple Word Radix-2 Montgomery Multiplication (MWR2MM) together with the quotient pipelining technique and proposed an architecture which can be mapped onto a modern high-performance DSP-oriented FPGA structure. We use a Montgomery multiplication algorithm based on quotient pipelining technique developed by Orup in 1995 [16].

III. Mathematical Background

Optimal ate pairing is an efficiently computable bilinear map $e : \mathbb{G}_2 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1 = E(\text{GF}(p))[n] \cap Ker(\pi_p - [1])$ and $\mathbb{G}_2 = E(\text{GF}(p^{12}) \cap Ker(\pi_p - [p])$ are additive groups, and $\mathbb{G}_T$ is a subgroup of the multiplicative group $\text{GF}^*(p^k)$. In this notation, $E(GF(p^k))[n]$ denotes points of an elliptic curve over $GF(p^k)$ of order n, π_p denotes the Frobenius endomorphism given as $(x, y) \mapsto (x^p, y^p)$. Barreto and Naehrig introduced a family of pairing-friendly elliptic curves defined by the following equation $E : y^2 = x^3 + b$ over a prime field GF(p), where $b \neq 0$. The BN parameters are defined by a suitable $z \in \mathbb{Z}$ such that $p = 36z^4+36z^3+24z^2+6z+1$, and n is the large odd prime dividing the curve order of $E(GF(p))$, given by the equation $n = 36z^4 + 36z^3 + 18z^2 + 6z + 1$. The embedding degree is defined as k such that $n|p^k - 1$.

At CRYPTO'16, Kim and Barbulescu presented an efficient number field sieve algorithm to solve discrete logarithm problem in a finite field [17]. As the hardness of solving DLP determines the security achieved by a pairing-based algorithm, the parameters earlier used for 126-bit security now provide approximately 100-bit security [17], [18] and [19]. We choose to focus on parameters to compute an optimal ate pairing for 100-bit security as most of the designs from the literature are reported using the same parameters [9], [10], [11], [12], [13]. However, our methodology can be applied to a different parameter set to target higher security levels as well. In order to achieve the 100-bit security level, we choose $z = -(2^{62} + 2^{55} + 1) < 0$. The exact computations involved in calculating optimal ate pairing are summarized in Algorithm 1 [10].

IV. Design Methodology

Our design methodology is shown in Fig. 1. The main elements of this methodology are the scheduler and the execution unit.

The operation of the system is verified using a generic testbench, written manually, comparing the actual hardware outputs with the expected outputs, generated using an equivalent reference software implementation based on the RELIC library [20].

A. Execution Unit

Our cryptoprocessor is composed of the Arithmetic Unit (AU) (containing several multipliers (MULT_0, MULT_1, ..., MULT_{r-1}) and an adder/subtractor (A/S)), a Local Memory

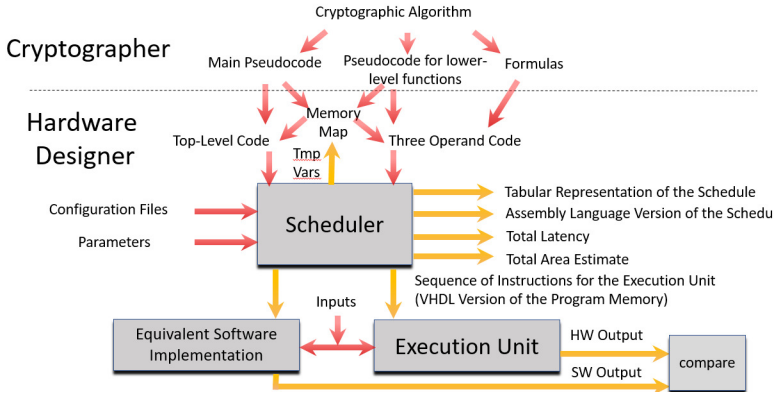

Fig. 1: Design Methodology

Algorithm 1 Optimal ate pairing over BN Curves [10]

Input: $P = (x_P, y_P) \in \mathbb{G}_1$, $Q = (x_Q\gamma^2, y_Q\gamma^3) \in \mathbb{G}_2$, with x_Q and $y_Q \in GF(p^2)$, $r = |6z+2| = \sum_{i=0}^{s-1} r_i 2^i$
Output: $a_{opt}(Q,P) \in \mathbb{G}_T$
$T \leftarrow (x_Q\gamma^2, y_Q\gamma^3, 1), f \leftarrow 1$
for i from $s-2$ downto 0 **do**
  $g \leftarrow l_{(T,T)}(P), T \leftarrow 2T, f \leftarrow f^2, f \leftarrow f \cdot g$
  if $(r_i = 1)$ **then**
    $g \leftarrow l_{(T,Q)}(P), T \leftarrow T + Q, f \leftarrow f \cdot g$
  end if
end for
$T \leftarrow -T, f \leftarrow f^{p^6}$
$Q_1 \leftarrow \pi_p(Q), Q_2 \leftarrow -\pi_p^2(Q)$
$g \leftarrow l_{(T,Q_1)}(P), T \leftarrow T + Q_1, f \leftarrow f \cdot g$
$g \leftarrow l_{(T,Q_2)}(P), T \leftarrow T + Q_2, f \leftarrow f \cdot g$
$f \leftarrow \left(f^{p^6-1}\right)^{p^2+1}$
$f \leftarrow f^{(p^4-p^2+1)/n}$
return f

Module, an instruction memory, global control unit (GCU), an input storage, and an output storage. Two primary components of the Execution Unit, Arithmetic Unit and Local Memory Module, are shown in Figure 2. The scheduler implemented using a Python script yields a sequence of instructions in the form of a VHDL ROM file. This instruction sequence is assumed to be preloaded into a program memory implemented as a ROM which are dispatched to the Arithmetic Unit by the Instruction Decoder.

The cryptoprocessor executes a sequence of multiple additions during one multiplication. To exploit parallelism even further, several multipliers are instantiated in the Arithmetic Unit to perform independent multiplications in parallel. The scheduler can be used to determine an optimal number of multipliers required to achieve either minimum execution time or minimum execution time·area product.

According to the local memory map shown in Table II, all inputs and parameters required to perform a pairing operation are loaded into the local memory initially. As the control unit reads instructions from the program memory, data is loaded from the local memory to the multiplier units one by one. Only after all operands are loaded into the multiplier units, the execution starts. As the local memory is dual-port, both operands of the multiplication or addition/subtraction can be read in one clock cycle. As all multiplications required in a batch start execution at the same time, we can use a single control unit for these operations. To utilize the architecture to its fullest, once all multipliers are loaded, and multiplications started, we perform all additions/subtractions that are part of a batch of instructions. Registers are required at the output of each multiplier to store partial results, as the local memory can support write operation only for one multiplier's results.

The internal datapath width is a multiple of $2k$ bits, as each Orup multiplier operates on $2k$-bit words at a time. In our modular multiplier design based on [14], [15], α is the number of $2k$ words, where k is 17. For 256-bit operands, α is 9, whereas for 512-bit operands α is 17. For a 256-bit Orup multiplier, after 42 clock cycles the multiplier starts generating k-bit words in each of the following $2*\alpha$ clock cycles to compute the final result.

The overall number of modular multiplications required for any pairing computation is quite large. Therefore, we exploit parallelism to execute these operations in the form of batches by instantiating multiple multiplier cores. By utilizing DSP units and BRAMs to construct these multipliers, the available embedded resources in FPGAs can be used instead of LUTs to exploit parallelism and still have a balanced resource utilization of the device being used.

Instruction Set Architecture: In our design, the instructions are stored in the ROM in the form of batches. Each batch can be composed of modular multiplications and modular additions/subtractions. The instructions for these operations, including the instruction called Batch Begin (BB), are categorized as arithmetic instructions in our instruction set architecture. The other set of instructions are called control flow instructions. We use them for the operations required to execute the top-level code. The input to the scheduler can be a three-operand file describing an arbitrarily-long sequence of multiplications, additions, and subtractions that can be scheduled statically, before run time, e.g., elliptic curve point addition or point doubling. The instructions are stored in the ROM in the form

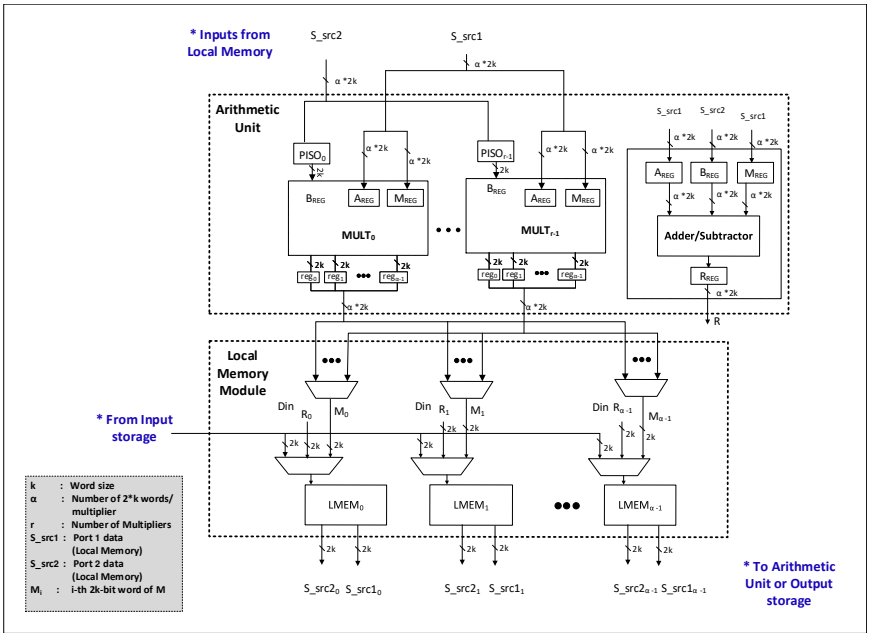

Fig. 2: Datapath of the Execution unit

of batches. Each batch starts with a Batch Begin instruction to indicate the number of independent multiplications and additions/subtractions that can be executed concurrently. The instructions stored in the instruction ROM are decoded by an instruction decoder which is part of the global control unit. The decoded information contains the addresses of operands and result, and also the control signals for the arithmetic unit. All the inputs and parameters are loaded into the Local Memory Unit before the computations start. After that instructions are read from the memory with a priority assigned to independent multiplications. During the execution of a multiplication, modular additions/subtractions are performed and their results are also written back to the Local Memory Module.

B. Scheduler of Field Operations for Pairing

Higher-level operations of pairing based cryptosystems, such as elliptic curve point addition, point doubling, field inversion, Miller Loop, and Final Exponentiation, are all composed of a sequence of field multiplications, squarings, additions and subtractions. We support a function-based input format, required to represent higher-level operations. This allows us to develop functions for each higher-level operation that can be flattened by our tools into a three-operand code enabling the automated scheduling of operations described using classical arithmetic notation. The main top-level algorithm, e.g., optimal ate pairing, is described in the form of the assembly language top-level code. This code is interpreted by the controller of the execution unit. It allows handling of conditional statements and loops from the main algorithm. This code is specific to each algorithm. However, its development time is relatively small, as only a few top-level operations, e.g., those shown in Algorithm 1, need to be described using this method.

At the second level, we have operations in some extension fields, as per the specification of the algorithm implemented as functions. We can take the top-level description, plug in macros for different intermediate-level operations, and unroll the entire description into a three-operand code composed of only modular multiplications and modular additions/subtractions. The three-operand code input is basically a set of operations described as a sequence of multiplications, additions and subtractions. On the left side there is a name of the output variable and on the right side there is a description of a two-operand operation.

For this effort, we have developed an automated scheduler using some concepts pioneered for the development of Application Specific Instruction Processors (ASIPs), which will accept as input a three-operand code (op3) file, either downloaded from the existing database, such as the Hyperelliptic.org Explicit Formulas Database [21], or developed from scratch. The scheduler accepts the following additional parameters:

- Number of multipliers: the number of modular multipliers available in the execution unit.
- Assembly file: the name of the assembly file to which the scheduler will write the assembly language version of the program.
- VHDL ROM file: the name of the VHDL file to which the scheduler will write the machine language version of the program.
- CSV file: the name of the CSV file to which a visual representation of the schedule will be written.

The scheduler will provide a detailed output, which will contain a description of a tree representation of the three-operand code (in a format that can be supported by phpSyntaxTree – a visual display tool [22]) and a textual description, containing every step in which a multiplication, addition, or subtraction will occur.

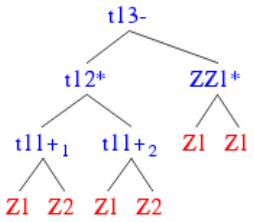

Fig. 3: Tree representation of the operations

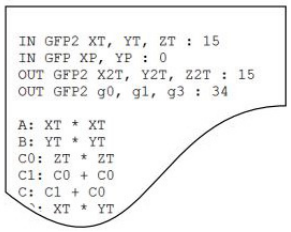

Fig. 4: Three Operand Code for Point Doubling and and line function computation

Detailed operation of the scheduler: First, the scheduler will process the op3 file, and develop a representation of the code in a data acyclic graphs (DAGs) as described in [23]. Each node represents the result of a given operation. In the current iteration, some of the techniques used to identify matching patterns within the DAGs are used, specifically ensuring that multi-input multiplication and addition can be performed in the smallest number of commands to prevent re-use. Once the representation is finalized, the scheduler builds a simulation of the running system–as discussed in [24]–to construct the optimum sequence of instructions, the scheduler develops three lists:

- Operations to be completed: a list containing the operation node, the maximum number of additions between the node and a root node, the maximum number of multiplications between the node and a root node. This list is sorted in descending order according to the number of remaining multiplications necessary to reach a root node. In case the number of multiplications is the same, the nodes with the largest number of remaining additions are listed first.
- Operations that have been completed: a list containing values that have already been calculated by the scheduled operations (when the scheduler launches, this list only contains the labels for inputs).
- Registers: a list that represents the memory entries for each value that is still used in the remaining nodes of the tree.

With the initial setup complete, the scheduler performs an Algorithm as defined in 2, which includes updates over the version identified in a previous version of the paper [25]. This algorithm is intended both to mimic the manual techniques used by the authors to develop schedules for any cryptographic formula and simulate the operation of the cryptographic coprocessor. Where manually optimizing a schedule for an elliptic curve sequence of basic Galois Field operations can take minutes or hours, the scheduler can generate the equivalent result within milliseconds or seconds. The scheduler aims to maximize utilization of the processor while performing multiplications. In particular, the schedule tracks the additions, subtractions, and moves that can be performed while multipliers are in use, to minimize the number of clock cycles when multipliers are idle.

C. Local Memory Map

The local memory map of our pairing coprocessor is shown in Table II. The coordinates of point P and point Q are

TABLE II: Local Memory and Register Map.

Local Memory Map		
Location in LMEM	Variable	Description
0	Xp:Xp*R	Xp:Xp in Montgomery Domain
1	Yp:Yp*R	Yp:Yp in Montgomery Domain
2	Xq_0:Xq_0*R	Xq_0:Xq_0 in Montgomery Domain
3	Xq_1:Xq_1*R	Xq_1:Xq_1 in Montgomery Domain
4	Yq_0:Yq_0*R	Yq_0:Yq_0 in Montgomery Domain
5	Yq_1:Yq_1*R	Yq_1:Yq_1 in Montgomery Domain
6	Zq_0:Zq_0*R = 1:R	Zq_0:Zq_0 in Montgomery Domain
7	Zq_1:Zq_1*R = 1:R	Zq_1:Zq_1 in Montgomery Domain
8	M	Modulus
Lines 9-52 ommitted for brevity		
Register Map		
R0	S	ceil($\log_2$r)-2
R1	N	ceil($\log_2$(-2t))-2

stored in the local memory initially. As Q is described using projective coordinates, and x_Q, y_Q, and z_Q are elements of $GF(p^2)$, each of the three coordinates is expressed as an element in the quadratic field. Rsq is required to convert the operands into Montgomery domain [26], and is stored in the local memory location 10. The value of $\tilde{M}$ is required for the operation of Orup Montgomery Multiplication [16], [14], [15]. It is computed only once as part of the pre-processing and used throughout the entire operation. The points P and Q are elements of groups G_1 and G_2, respectively. Optimal ate pairing maps them to an element in the extension field $GF(p^{12})$. Locations 21-33 are reserved to store the local variable f required in pairing. We also reuse these locations during the optimal ate processing. Additional memory locations are required to store the results of the Frobenius maps. The sparse element g requires only 6 memory locations as the other $GF(p)$ components are equal to zero. During the hard part of the final exponentiation, an inversion in $GF(p^{12})$ is required three times. The exponents for these inversions are also stored in the local memory at locations 12, 13, and 14.

D. Assembly Language Version of the Program

The first step is to convert operands in normal domain to operands in Montgomery domain [26]. These operands are converted back to normal domain only after the entire optimal ate pairing is completed. We convert the coordinates of both points P and Q to Montgomery domain. After that the coordinates of point Q are copied into the temporary memory locations for a variable T, and later used in the Miller loop. According to Algorithm 1 for optimal ate pairing, line 5 requires the computation of line function, point doubling

Algorithm 2 Automated Scheduler

Input: Any OP3 Code in tree form and a list containing each node in the tree that represents the list of operations to be completed
Output: Efficient scheduler for the formula
Pre-processing: The list of operations is sorted by multiplication depth–the maximum number of multiplications from each node to a root. The list is sorted in this order.
while Number of operations to be completed > 0 **do**
    IdealMults $\leftarrow$ the multiplication depth of the first N multiplications, where N is the number of multipliers
    onlyAdders $\leftarrow$ the list of adders whose operands have been calculated that have an adder as a parent
    waitingMultipliers $\leftarrow$ the first N multiplications whose operands have been calculated
    if the multiplication depth of waitingMultipliers and idealMults are the same **then**
        IsIdeal$\leftarrow$ True
    else
        IsIdeal $\leftarrow$ False
    end if
    if Multipliers are not in use **then**
        if The size of waitingMulitpliers $< N$ and $<$ the number of remaining possible multiplications **then**
            pauseMultipliers $\leftarrow$ True
        else if The size of onlyAdders > 0 and not IsIdeal **then**
            pauseMultipliers $\leftarrow$ True
        else
            pauseMultipliers $\leftarrow$ False
        end if
    end if
    if pauseMultipliers **then**
        Append the first node in onlyAdders to the list to be performed
    else
        for For each node in the list of possible operations **do**
            if this node is a multiplication and multiplication is not occurring and the node operands have been calculated **then**
                Append this node to the list to be processed unless there would be too many simultaneous multiplications
            else if this node is an addition or subtraction and the node operands have been calculated **then**
                if $(Tadd \cdot (\#addstoprocess + 1)) + (Tmov \cdot (\#movstoprocess + 1)) \geq Tmul$ **then**
                    Break
                else
                    Append this node to the list to be performed
                end if
        end for
    end if
    Increment the simulated clock and process all operations to be processed in parallel
    Add all nodes that have been completed to the completed list. Record the starting and ending time.
    Determine if existing registry entries are used by any remaining operations
    Determine if a new registry entry is needed and add if necessary
    Assign the outputs of completed operations to registry entries.
loop

and squaring in GF(p^{12}). These operations do not have any dependencies between each other, so we combine them in one hierarchical input file called PP_LN_DBL_SQR. Once these operations are performed, a sparse multiplication called PP_FG is performed. The element f in this multiplication is dense, however g is a sparse element. The aforementioned two operations are always performed inside of a Miller loop.

E. Pseudocode of the Lower Level Functions

Three Operand Code: The three-operand code in case of pairing involves a hierarchy of operations. For Algorithm 3, the lower level functions required to perform the operation are multiplication, squaring, addition, negation in $GF(p^2)$ and multiplication by the quadratic non-residue i. The three-operand code shown in Figure 4 performs the same operations as those shown in Algorithm 3. The header of the file includes elements in the base field $GF(p)$ and the extension field $GF(p^2)$. Each element in $GF(p^2)$ requires two memory locations in the local memory of our execution unit. The

Algorithm 3 Doubling Step and Line Function

Input: $P = (x_P, y_P) \in E(\text{GF}(p)); T = (X_T\gamma^2, Y_T\gamma^3, Z_T) \in E(\text{GF}(p^{12}))$ with X_T, Y_T and $Z_T \in \text{GF}(p^2)$
Output: $2T, l_{(T,T)}(P)$
$B \leftarrow Y_T^2, E \leftarrow 2Y_TZ_T, C \leftarrow 3Z_T^2, D \leftarrow 2X_TY_T$
$A \leftarrow X_T^2, H \leftarrow 3C$
$F \leftarrow B + iH, G \leftarrow B - iH, J \leftarrow 4HC,$
$g_0 \leftarrow Ey_P, g_1 \leftarrow -3Ax_P, g_3 \leftarrow B + iC,$
$I \leftarrow G^2, Z_{2T} \leftarrow 4BE,$
$X_{2T} \leftarrow DF, Y_{2T} \leftarrow I + J$
return $(X_{2T}\gamma^2, Y_{2T}\gamma^3, Z_{2T}), g_0, g_1, g_3$

scheduler includes a utility to flatten the hierarchical three-operand code to a simple three-operand format with elements only in $GF(p)$.

A number of higher level algorithms have functions that operate on different types of data and different variables. To this end, we developed a preprocessor and a function definition language to support writing a high level function, such as optimal ate pairing, that is then converted into three-operand code. The preprocessor follows a model similar to the original C++ compiler, which takes C++ code and converts it to C code using . The preprocesser takes the higher level pairing and elliptic curve functions and converts them into three-operand codes for operations in $GF(p)$. The functions are written in three-operand code with a header that defines the information within the function:

```
*, GFP2, GFP2
IN GFP2 (a0, a1), (b0, b1)
OUT GFP2 (c0, c1)
v0: a0 * b0
v1: a1 * b1
c0: v0 + v1
```

This function describes an operation among two variables of type GFP2. Each of the operands is defined as a tuple consisting of two variables in $GF(p)$, with the results being stored in a tuple consisting of c0 and c1.

The preprocessor uses macro expansion, as described in [23] to track variables' value and any operations defined in the function folder and convert them to the lower-level three-operand code supported by the scheduler.

V. Results and Their Analysis

The scheduler has been tested first using medium complexity functions, such as elliptic curve point addition and point doubling. Automatically generated results matched exactly the best results obtained by the authors using manual optimizations, both in terms of the execution time and the number of memory locations required [25].

Using the scheduler, we were able to identify how effective multiple multipliers will be in implementing various steps of optimal ate pairing. The shortest (ideal) execution time (expressed in clock cycles) is given by

$$CC_{ideal} = \lfloor \#muls/Nm \rfloor \cdot CC(Nm) + CC(\#muls \bmod Nm), \quad (1)$$

TABLE III: Features of the execution unit for the operand sizes of 256 bits.

Number of clock cycles per multiplication, $Mcomp$	60
Number of clock cycles per add/sub, $AScomp$	5
Number of clock cycles per move, $CMove$	1
Time to read data, Mrd	1
Time to write data, Mwr	1

TABLE IV: The statistics information about the selected functions of optimal ate pairing.

Function	#calls	#mul calls	#add sub calls	#mov calls	total #muls	%muls
pp_ln_dbl_sq	107	84	262	315	8988	58.11
pp_fg	113	48	122	135	5424	35.07
pp_ln_add	4	56	83	69	224	1.45
pp_neg	1	0	2	2	0	0.00
pp_inv	1	0	6	6	0	0.00
pp_frb1	1	8	12	12	8	0.05
pp_frb2	1	2	2	7	2	0.01
pp_ln_add_q1	1	56	83	69	56	0.36
pp_ln_add_q2	1	56	83	69	56	0.36
pp_fexp1	1	364	912	898	364	2.35
pp_fexp2	1	345	1789	500	345	2.23

where $\#muls$ is the total number of multiplications, Nm is the number of multipliers, and $CC(i)$ is the number of clock cycles required to execute a batch including i multiplications. $CC(i) = i \cdot Mrd + Mcomp + i \cdot Mwr$, where Mrd, $Mcomp$, and Mwr are defined in Table III.

The Eq. (1) assumes that for all multiplication time slots, except possibly the last one, Nm multiplications can be scheduled in parallel, and all remaining operations (additions, subtractions, and moves), can be executed in parallel to multiplications (and thus do not affect the execution time). In practice, these assumptions are not likely to be true, because of dependencies between multiplications, and because the number of additions/subtractions that can be performed in parallel to i simultaneous multiplications is limited by the formula $Tadd(j) \leq Tmul(i)$, where the time of i simultaneous multiplications is given by $Tmul(i) = i \cdot 1 + 60 + i \cdot 1 = 60 + 2i$ for all $i \leq Nm$, and the time of j sequential additions/subtractions is given by $Tadd(j) = j \cdot (1 + 5 + 1) = 7 \cdot j$.

Thus, $7 \cdot j \leq 60 + 2i$ resulting in $j \leq 8.57 + 0.33i$. Since i can be as small as 1 and as large as Nm, the current version of the scheduler uses the value of j for which the dependence holds even for the minimum possible value of i, namely $j = 8$.

The overhead of actual computations can be defined as

$$\text{Overhead} = (\text{CC}_{actual} - \text{CC}_{ideal})/\text{CC}_{ideal} \cdot 100\%. \quad (2)$$

The above definitions and formulas are specific to the execution unit characterized in Table III.

To illustrate the impact of various numbers of multipliers, we analyzed the execution time of three functions within the optimal ate pairing algorithm, with different numbers of moves, multiplications, and additions, as characterized in Table IV. By running the scheduler for each function using multiple values of Nm, we can determine which values of Nm are optimal from the point of view of the computation time, t=CC_{actual}, and the product of the computation time and area

TABLE V: Results for three representative functions. The minimum execution times, t, and the minimum time-area products, t·area, are shown in bold.

	$f \leftarrow (f^{p^6-1})^{p^2+1}$ PP_FEXP1			$f \leftarrow f^{(p^4-p^2+1)/n}$ PP_FEXP2			$f \cdot g$ PP_FG			area [LUTs]
Nm	Over-head	t [cycles]	t·area [cycles·LUTs]	Over-head	t [cycles]	t·area [cycles·LUTs]	Over-head	t [cycles]	t·area [cycles·LUTs]	
1	0	22665	$39.0 \cdot 10^6$	0	7222	$12.4 \cdot 10^6$	3	3058	$5.3 \cdot 10^6$	1719
2	3	11958	$\mathbf{32.4 \cdot 10^6}$	14	4228	$\mathbf{11.5 \cdot 10^6}$	15	1768	$\mathbf{4.8 \cdot 10^6}$	2712
3	20	9679	$35.9 \cdot 10^6$	60	4121	$15.3 \cdot 10^6$	28	1353	$5.0 \cdot 10^6$	3705
4	57	9685	$45.5 \cdot 10^6$	106	4069	$19.1 \cdot 10^6$	42	1161	$5.5 \cdot 10^6$	4698
5	80	9199	$52.4 \cdot 10^6$	134	3920	$22.3 \cdot 10^6$	62	1128	$6.4 \cdot 10^6$	5691
6	107	9076	$60.7 \cdot 10^6$	175	3934	$26.3 \cdot 10^6$	98	1138	$7.6 \cdot 10^6$	6684
7	135	9036	$69.4 \cdot 10^6$	209	3864	$29.7 \cdot 10^6$	117	1120	$8.6 \cdot 10^6$	7677
8	155	8900	$77.2 \cdot 10^6$	243	3884	$33.7 \cdot 10^6$	145	1115	$9.7 \cdot 10^6$	8670
9	180	8912	$86.1 \cdot 10^6$	289	3932	$38.0 \cdot 10^6$	145	1117	$10.8 \cdot 10^6$	9663
10	197	8748	$93.2 \cdot 10^6$	311	3917	$41.7 \cdot 10^6$	177	1098	$11.7 \cdot 10^6$	10656
11	213	8657	$100.8 \cdot 10^6$	331	**3842**	$44.8 \cdot 10^6$	177	**1096**	$12.8 \cdot 10^6$	11649
12	232	**8604**	$108.8 \cdot 10^6$	370	3914	$49.5 \cdot 10^6$	226	**1096**	$13.9 \cdot 10^6$	12642

TABLE VI: Results for the entire optimal ate pairing operation. The minimum execution times, t, and the minimum time-area products, t·area, are shown in bold.

Nm	Over-head	t [cycles]	t·area [cycles·LUTs]	area [LUTs]	Speed up
1	1%	972,549	$1678.1 \cdot 10^6$	1719	1.00
2	10%	546,363	$\mathbf{1481.7 \cdot 10^6}$	2712	1.78
3	40%	475,150	$1760.4 \cdot 10^6$	3705	2.05
4	63%	427,753	$2009.6 \cdot 10^6$	4698	2.27
5	96%	424,229	$2414.3 \cdot 10^6$	5691	2.29
6	124%	416,059	$2780.9 \cdot 10^6$	6684	2.34
7	149%	407,597	$3129.1 \cdot 10^6$	7677	2.39
8	178%	408,904	$3545.2 \cdot 10^6$	8670	2.38
9	201%	403,952	$3903.4 \cdot 10^6$	9663	2.41
10	228%	405,920	$4325.5 \cdot 10^6$	10656	2.40
11	247%	$\mathbf{399,837}$	$4657.7 \cdot 10^6$	11649	2.43
12	274%	403,993	$5119.9 \cdot 10^6$	12642	2.41

TABLE VII: Comparison of this work with previous implementations of optimal ate pairing at the 100-bit security level.

Reference	Resources [Slices, DSPs]	Freq [MHz]	Cycles x10^3	Time [ms]	Time·Area [ms·Slices]
This work	1636, 36	227	546	2.405	3935
[9]	5237, 64	210	78	0.338	1770
[10]	5163, 144	166	62	0.375	1926
[13]	5976, 30	145	80	0.552	3299
[11]	7032, 32	250	143	0.572	4022
[12]	4014, 42	210	245	1.167	4684

of the execution unit, t·area. Performing this calculation for all values of Nm in the range from 1 to 12 for the final exponentiation step in Algorithm 1, on the t2.medium instance of the Amazon Web Services (AWS) Elastic Compute Cloud (EC2), took approximately 30 minutes, with all remaining functions in the algorithm able to be completed in less time.

The area of the execution unit (expressed in LUTs) for a different number of multipliers has been calculated based on the post-place & route results obtained for Nm=1, 2, and 4, in case of the Xilinx Zynq-7000 xc7z020clg484-1 device. The results for the aforementioned three cases were generated using Vivado Design Suite 2016.2. The extrapolated numbers of LUTs for all investigated values of Nm are shown in the rightmost column of Table V. In our analysis, we assume that any dependence of the clock period on the number of multipliers is negligible (as the critical path is located inside of each multiplier), and thus, the number of clock cycles can be used as a proper measure of the execution time, across different values of Nm. This assumption has been verified by implementing the execution unit using the maximum number of multipliers, $Nm = 12$, limited by the number of DSP units, available in the selected Zynq-7000 device.

As shown in Table V, for the selected three functions characterized in Table IV, simply adding new multiplier cores does not necessarily improve the total runtime of the function. The function $f \leftarrow (f^{p^6-1})^{p^2+1}$ gets the best performance with 12 multipliers, while the function $f \leftarrow f^{(p^4-p^2+1)/n}$ gets the best performance with 11 multipliers, and $f \cdot g$ gets the best performance with 11 or more multipliers. While for some applications, the designers may choose to go with pure performance, t, for others it may be preferable to use the time·area product, t·area. For each function, two multipliers gave the smallest time·area product.

We subsequently calculated t and t·area for each step in the optimal ate pairing, as described in Algorithm 1 and shown in Table VI. To calculate t for the Miller loop, we used a value of $s = 108$, derived based on the values of z and r. This means the number of doublings is $s - 1 = 107$. Based on the algorithm, the number of additions is equal to the number of 1s in the binary representation of r, without counting the most significant bit of r, which is 4. Additionally, the total number of memory locations required stayed relatively constant regardless of the choice of Nm. $Nm = 1$ required 98 memory locations and $Nm = 1..12$ required 99 memory locations. These include the 52 memory locations shown in Table II.

Although obtaining the most efficient design was not our primary goal, in Table VII, we compare our semi-automatically-generated implementation of optimal ate pairing with several previously reported manually-scheduled designs, providing exactly the same or equivalent functionality, at the same security level. All previous designs have been implemented using Virtex-6 FPGAs. Our design has been

implemented using Zynq-7000 family (with the very similar Slice and DSP unit architecture).

Our design, based on two Montgomery multipliers, has the smallest resource utilization in terms of Slices. It has also the fourth best product Time·Area, somewhat surprisingly even smaller than the design by Cheung et al. [11] employing RNS and several other optimization techniques. The longer execution time is primarily the result of using much simpler execution unit based on the Montgomery multiplication in $GF(p)$, leading to significantly larger number of clock cycles per modular multiplication. On the other hand, the longer execution time is clearly not the effect of a sub-optimal scheduling, as our semi-automatically generated schedule has an overhead of only 10% compared to the ideal case for a given execution unit with the number of multipliers, Nm=2. In order to successfully compete with the designs from [9], [10], [13], the execution unit and scheduler would need to support additional optimizations, such as the use of RNS, lazy reductions, pipelining, Karatsuba or Toom-Cook-Karatsuba multiplication, multiplication by a constant, multiplicative inverse unit, etc.

VI. Conclusions and Future Work

We have developed the methodology for choosing an optimal number of multipliers in the general-purpose execution unit used to perform algorithms based on prime field arithmetic. We have applied our methodology to the specific case of optimal ate pairing. Using the scheduler, our framework can support any elliptic curve or paring-based cryptographic algorithms that can be described using three-operand code.

Future work will include extending the scheduler to support multiple different types of algorithms and execution units as well as some more modern techniques for performing pattern analysis in the DAG and modifying optimization algorithms to allow for different targets. In particular, we intend to provide support for using different hardware architectures to perform modular multiplication. These architectures may provide features such as the use of RNS, lazy reductions, Karatsuba multiplication, etc. Introducing a pipelined adder/-subtractor in the execution unit can also lead to processing multiple additions/subtractions in parallel. However, it requires the scheduler support to handle dependencies between the operands required for these additions. Additionally, a specialized hardware architecture to perform a modular multiplicative inversion using extended Euclidean algorithm will be developed to reduce the overall execution time of the design, and the scheduler extended to support this operation.

Acknowledgment

The authors would like to acknowledge the U.S. Department of Commerce (NIST) for sponsoring this effort.

References

[1] D. Boneh and M. K. Franklin, "Identity-based encryption from the Weil pairing," in *21st Annual International Cryptology Conference, CRYPTO*. Springer, 2001, pp. 213–229.

[2] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *7th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT*. Springer, 2001, pp. 514–532.

[3] D. Boneh and B. Xavier, "Secure identity based encryption without random oracles," in *24th Annual International Cryptology Conference, CRYPTO*, 2004.

[4] A. Joux, "A one round protocol for tripartite Diffie-Hellman," in *4th International Symposium on Algorithmic Number Theory*, ser. ANTS-IV. Springer, 2000, pp. 385–394.

[5] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," *Discrete Appl. Math.*, vol. 156, no. 16, pp. 3113–3121, Sep. 2008.

[6] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted Edwards curves," in *Cryptology in Africa, 1st International Conference on Progress in Cryptology, AFRICACRYPT*. Springer, 2008, pp. 389–405. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788634.1788672

[7] H. Gu, D. Gu, and W. Xie, "Efficient pairing computation on elliptic curves in Hessian form," in *13th International Conference on Information Security and Cryptology, ICISC*. Springer, 2011, pp. 169–176.

[8] S. Ghosh, A. Kumar, A. Das, and I. Verbauwhede, "On the implementation of unified arithmetic on binary Huff curves," *IACR Cryptology ePrint Archive*, vol. 2015, p. 423, 2015.

[9] G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede, "Faster pairing coprocessor architecture," in *Pairing-Based Cryptography - Pairing 2012 - 5th International Conference, Cologne, Germany, May 16-18, 2012*, 2012, pp. 160–176.

[10] S. Ghosh, I. Verbauwhede, and D. Roychowdhury, "Core based architecture to speed up optimal ate pairing on FPGA platform," in *Pairing-Based Cryptography – Pairing 2012*. Springer, 2013, pp. 141–159.

[11] R. C. C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. Yao, "FPGA implementation of pairings using residue number system and lazy reduction," in *13th International Conference on Cryptographic Hardware and Embedded Systems, CHES*. Springer, 2011, pp. 421–441.

[12] J. Fan, F. Vercauteren, and I. Verbauwhede, "Efficient hardware implementation of Fp-arithmetic for pairing-friendly curves," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 676–685, May 2012.

[13] A. Sghaier, L. Ghammam, Z. Medien, S. Duquesne, and M. Machhout, "Area-efficient hardware implementation of the optimal Ate pairing over BN curves," *IACR Cryptology ePrint Archive*, 2015.

[14] D. Suzuki, "How to maximize the potential of FPGA resources for modular exponentiation," in *Workshop on Cryptographic Hardware and Embedded Systems—CHES*. Springer, 2007, pp. 272–288.

[15] D. Suzuki and T. Matsumoto, "How to maximize the potential of FPGA-based DSPs for modular exponentiation," *IEICE Transactions*, vol. 94-A, no. 1, pp. 211–222, 2011.

[16] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *12th Symposium on Computer Arithmetic*, Jul 1995, pp. 193–199.

[17] T. Kim and R. Barbulescu, "Extended tower number field sieve: A new complexity for the medium prime case," in *36th Annual International Cryptology Conference – CRYPTO*. Springer, 2016, pp. 543–571.

[18] T. Kim and J. Jeong, "Extended tower number field sieve with application to finite fields of arbitrary composite extension degree," in *Public-Key Cryptography – PKC*. Springer, 2017, pp. 388–408.

[19] P. Sarkar and S. Singh, "A general polynomial selection method and new asymptotic complexities for the tower number field sieve algorithm," in *Advances in Cryptology – ASIACRYPT*. Springer, 2016, pp. 37–62.

[20] D. F. Aranha and C. P. L. Gouvêa, "RELIC is an Efficient LIbrary for Cryptography," http://code.google.com/p/relic-toolkit/.

[21] D. J. Bernstein and T. Lange, "Explicit-formulas database," http://www.hyperelliptic.org/EFD.

[22] "phpSyntaxTree," http://ironcreek.net/phpsyntaxtree/.

[23] G. H. Blindell, "Universal instruction selection," April 2018.

[24] M. K. Jain, M. Balakrishnan, and A. Kumar, "Asip design methodologies: Survey and issues," in *VLSI Design, 2001. Fourteenth International Conference on*. IEEE, 2001, pp. 76–81.

[25] R. Shahid, T. Winograd, and K. Gaj, "A generic approach to the development of coprocessors for elliptic curve cryptosystems," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 158–167.

[26] P. Montgomery, "Modular multiplication without trial division," *Math. Comp.*, vol. 44, no. 170, pp. 519–521, 1985.