Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 439021, 14 pages
doi:10.1155/2012/439021

Research Article

A Hardware-Accelerated ECDLP with High-Performance Modular Multiplication

Lyndon Judge, Suvarna Mane, and Patrick Schaumont

Bradley Department of Electrical and Computer Engineering, Center for Embedded Systems for Critical Applications (CESCA), Virginia Tech, Blacksburg, VA 24061, USA

Correspondence should be addressed to Lyndon Judge, lvjudge1@vt.edu

Received 4 May 2012; Accepted 17 September 2012

Academic Editor: René Cumplido

Elliptic curve cryptography (ECC) has become a popular public key cryptography standard. The security of ECC is due to the difficulty of solving the elliptic curve discrete logarithm problem (ECDLP). In this paper, we demonstrate a successful attack on ECC over prime field using the Pollard rho algorithm implemented on a hardware-software cointegrated platform. We propose a high-performance architecture for multiplication over prime field using specialized DSP blocks in the FPGA. We characterize this architecture by exploring the design space to determine the optimal integer basis for polynomial representation and we demonstrate an efficient mapping of this design to multiple standard prime field elliptic curves. We use the resulting modular multiplier to demonstrate low-latency multiplications for curves secp112r1 and P-192. We apply our modular multiplier to implement a complete attack on secp112r1 using a Nallatech FSB-Compute platform with Virtex-5 FPGA. The measured performance of the resulting design is 114 cycles per Pollard rho step at 100 MHz, which gives 878 K iterations per second per ECC core. We extend this design to a multicore ECDLP implementation that achieves 14.05 M iterations per second with 16 parallel point addition cores.

1. Introduction

Elliptic curve cryptosystems (ECC), independently introduced by Miller [1] and Koblitz [2], have now found significant place in the academic literature, practical applications, and security standards. Their popularity is mainly because their shorter key sizes offer high levels of security relative to other public key cryptosystems, such as RSA. The security of ECC relies on the difficulty of elliptic curve discrete logarithmic problem (ECDLP) [3]. By definition, ECDLP is to find an integer n for two points P and Q on an elliptic curve E defined over a finite field $\mathbb{F}_q$ such that

$$Q = [n]P. \tag{1}$$

Here, $[n]$ denotes the scalar multiplication with n.

The Pollard rho method [4] is the strongest known attack against ECC today. This method solves ECDLP by generating points on the curve iteratively using a pseudorandom iteration function $f : \langle S\rangle \rightarrow \langle S\rangle$ such that $X_{i+1} = f(X_i)$. Since the elliptic curve is defined over a finite field, $\langle S\rangle$ is finite and the walk will eventually encounter the same point twice resulting in a collision. When a collision occurs, the ECDLP is solved (for well-chosen form of X_c; see Section 3). Several optimizations to the Pollard rho method have been proposed to allow independent parallel walks [5], better iteration functions [6, 7], and more efficient collision detection [8].

There have been several different approaches to implement Pollard rho algorithm on software and hardware platforms. Most of the solutions are implemented on software platforms using general purpose workstations, such as clusters of PlayStation3 [9], Cell CPUs [10], and GPUs [11, 12]. These software approaches are inherently limited by the sequential nature of software on the target platform.

Programmable hardware platforms are an attractive alternative to the above because they efficiently support parallelization. However, most of the FPGA-based solutions that have been proposed do not deal well with the control

complexity of ECDLP. Instead, they focus on the efficient implementation of datapath operations and ignore the system integration aspect of the solution.

There has been little work in the area of supporting or accelerating a full Pollard rho algorithm on a hardware-software platform. Our solution, therefore, goes one step further as we demonstrate the parallelized Pollard rho algorithm on FPGA along with its integration to a software driver. We start from a reference software implementation, and demonstrate an efficient, parallel implementation of the prime field arithmetic for primes of the form $(2^n - m)/k$. We also present a novel high-performance architecture for modular multiplication that can be applied to a variety of standard prime field elliptic curves specified by the National Institute of Standards and Technology (NIST) [13] and Standards for Efficient Cryptography Group (SECG) [14].

Most existing ECDLP solutions for hardware platforms target curves over binary fields because the arithmetic is more hardware friendly. However, our work shows that implementation of prime field arithmetic on hardware can be as feasible as binary field arithmetic.

We use a computing platform by Nallatech, which consists of a quad-core Xeon core (E7310, 1.6 GHz) with a tightly coupled Virtex-5 (xq5vsx240t) FPGA. The hardware runs at 100MHz and uses 5229 slices per ECC core.

2. Related Work

The software solution proposed by Bernstein on Cell CPUs is the fastest existing software solution to the ECDLP for secp112r1 curve [15]. It uses the negation map and noninteger polynomial-basis arithmetic to report speedup over a similar solution by Bos et al. [9]. Both of these software solutions use prime field arithmetic in an affine coordinate system, and they exploit the SIMD architecture and rich instruction set of the Cell CPU. Another software solution by Bos [10] describes the implementation of parallel Pollard rho algorithm on synergistic processor units of cell broadband engine architecture to approach the ECC2K-130 Certicom challenge. High-performance ECDLP solutions for GPUs platforms have also been proposed in [12].

Fan et al. propose the use of a normal-basis, binary field multiplication to implement a high-speed attack on ECC2K-130 [16] using Spartan-3 FPGAs. Fan's solution outperforms attacks on the same curve using GPUs [12] and Cell CPUs [10], demonstrating the suitability of FPGA platforms for solving large ECDLPs in binary fields. Another binary field solution, for the COPACOBANA [17] FPGA cluster, targets a 160-bit curve [18]. Güneysu et al. propose an architecture to solve ECDLP over prime fields using FPGAs and analyze its estimated performance for different ECC curves ranging from 64-to 160-bit fields [19].

Among hardware-based solutions, Meurice de Dormale et al. propose an FPGA solution to attack the ECC Certicom challenge for GF(2^{113}) [20]. Though it discusses the hardware-software integration aspect of the solution, the authors did not confirm if their system was operational. The authors of [19] discuss some aspects of system-level integration for their prime-field ECDLP system. A three-layer hybrid distributed system is described by Majkowski et al. to solve ECDLP over binary field [21]. It uses the general purpose computers with FPGAs and integrates them with a main server at the top level.

The outline of the paper is as follows. In the next section, we discuss the background of the parallel Pollard rho algorithm. We present our high-performance architecture for modular multiplication and demonstrate its applicability to primes of the form $(2^n - m)/k$ in Section 4. In Section 5, we discuss the additional modular arithmetic units and we describe the complete hardware-software integrated system architecture in Section 6. Section 7 shows implementation results, including measured performance for secp112r1 ECDLP, and we conclude the paper in Section 8.

3. Pollard Rho Algorithm

Let p be a prime and $\mathbb{F}_p = \mathrm{GF}(p)$. Given the elliptic curve E over $\mathbb{F}_p$ of order $l = |E(\mathbb{F}_p)|$, let $S \in E(\mathbb{F}_p)$ be a point of order l. Solving the ECDLP requires finding an integer n given two points $P, Q \in \langle S \rangle$ such that $Q = nP$.

The Pollard rho algorithm [4] uses a pseudorandom iteration function $f : \langle S \rangle \rightarrow \langle S \rangle$ to solve the ECDLP. It conducts a pseudorandom walk by starting from a random seed point on the curve, $X_0 = a_0P + b_0Q$ for random $a_0, b_0 \in \mathbb{Z}$, and generating subsequent points using the iteration function $X_{i+1} = f(X_i)$. Since the elliptic curve is defined over a finite field, the iteration function will eventually produce the same point twice, resulting in a cycle.

The name of the algorithm, rho, expresses the Greek letter ρ, which shows a walk ending in a cycle. Cycles can be efficiently detected using Floyd's cycle-finding method or Brent's cycle-finding algorithm [8]. The collision point, which gives the solution to the ECDLP, is located at the starting point of the cycle. Therefore, the underlying idea of this algorithm is to search for two distinct points on the curve such that $f(X_i) = f(X_j)$. Due to the birthday paradox, assuming a random iteration function, the expected number of iterations to find collision is $\sqrt{(\pi \cdot |\langle X \rangle|)/2}$ [4].

The effectiveness of Pollard's rho method depends on the randomness of the iteration function. As such, studies have been conducted to evaluate the strength of various proposed iteration functions [6, 7, 22]. Teske proposes an additive iteration function:

$$f(X_i) = X_i + R_i(X_i), \tag{2}$$

where R_i is an index function $\langle S \rangle \rightarrow \{0, 1, \ldots, r-1\}$ and each element $R_j = a_jP + b_jQ$, for random values $a_j, b_j \in \mathbb{Z}$ [7]. Based on analysis by Teske [6, 7] and Bos et al. [22], the additive iteration function is more similar to a truly random walk than Pollard's original function and other proposed variants. Furthermore, an additive walk with $r \geq 16$ is very close to a true randomness and achieves speedup of 1.25X over Pollard's iteration function [6]. Thus, we perform an additive walk and choose $r = 16$.

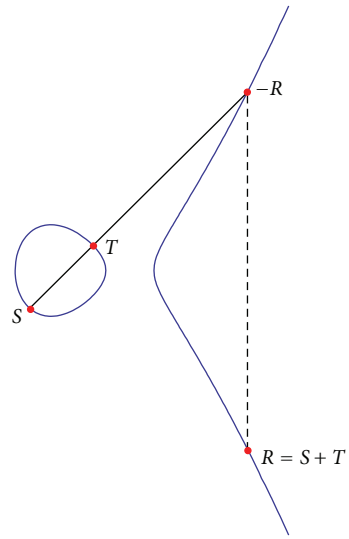

Figure 1: Geometric representation of elliptic curve point addition.

Using the additive walk, a collision occurs when two points are found such that

$$X_i + R_i = X_j + R_j. \tag{3}$$

Based on the definition of the index function R_i, the collision points can be rewritten as

$$c_i P + d_i Q = c_j P + d_j Q, \tag{4}$$

where $c_i = \sum_{k=0}^{i} a_k$ and $d_i = \sum_{k=0}^{i} b_k$. The solution can then be obtained as

$$n = \left[\frac{c_i - c_j}{d_j - d_i}\right] \text{mod} l. \tag{5}$$

3.1. Point Addition. Each iteration of the Pollard's rho algorithm requires a point addition of the current point, X_i, with a precomputed combination of P and Q, $R_i(X_i)$. Addition of two distinct points for an elliptic curve over a finite field GF(p), $S + T = R$ is defined geometrically as shown in Figure 1. To compute the addition, a line is drawn through the two points S and T. The line intersects exactly one other point of the elliptic curve. The intersection point is $-R$, which can be reflected across the x-axis to find R.

Algebraically, adding two points $S = (x_S, y_S)$ and $T = (x_T, y_T)$ gives the sum $R = (x_R, y_R)$, where

$$\begin{aligned} x_R &= \lambda^2 - (x_S + x_T), \\ y_R &= \lambda(x_S - x_R) - y_S, \\ \lambda &= \frac{y_S - y_T}{x_S - x_T}. \end{aligned} \tag{6}$$

Each point addition requires 2 multiplications modulo p, 1 modular squaring, 5 modular subtractions, 1 modular addition, and 1 modular inversion. The sequence of arithmetic operations for a point addition is shown in Table 1.

Table 1: Arithmetic operations for point addition: $S + T = R$.

Operation	Function performed
Modular subtraction	$t1 = y_S - y_T$
Modular subtraction	$t2 = x_S - x_T$
Modular inversion	$t2 = t2^{-1}$
Modular multiplication	$t4 = t2 \cdot t1 : \lambda$
Modular squaring	$t1 = t4 \cdot t4$
Modular addition	$t3 = x_S + x_T$
Modular subtraction	$t1 = t1 - t3 : x_R$
Modular subtraction	$t2 = x_R - t1$
Modular multiplication	$t3 = t2 \cdot t4$
Modular subtraction	$t3 = t3 - y_S : y_R$

By Fermat's Little Theorem, $z^{-1} = z^{p-2}$ for $z \in \text{GF}(p)$. Thus, computing modular inversion requires modular exponentiation, making inversion the most computationally expensive operation of the point addition. Optimization at the system level can reduce the cost of inversion through the use of Montgomery's trick [23]. Montgomery's trick makes it possible to share the cost of an inversion among M computations by performing inversions on vectors of M points simultaneously. Since the computational cost of inversion is two orders of magnitude more than other arithmetic operations, this results in significant savings over computation of M individual modular inversions. We apply Montgomery's trick by implementing a vectorized point addition datapath that computes M random walks on a single ECC core and shares the inversion cost across all walks.

Despite optimizations, inversion remains an expensive operation. Since exponentiation is achieved using multiplication, the cost of inversion is directly tied to the cost of modular multiplication. Therefore, design of an efficient modular multiplier is an important aspect of the system to maximize overall performance.

3.2. Comparison of Field Arithmetic in GF(p) and GF(2^m). Standards for elliptic curve cryptography have been defined for curves over both binary and prime fields [13, 14]. Binary fields can be represented with a polynomial basis or normal basis. In general, hardware implementations favor polynomial basis because it allows simplified reduction through the use of an irreducible polynomial of the form $x^{m-1} + \cdots + x^2 + x^1 + x^0$. Prior work has confirmed the idea that binary field curves are better suited for hardware platforms. A comparison of ECDLP engines by [20] shows a significant speedup for binary field ECDLP over prime field ECDLP implemented on FPGA platform. The performance discrepancy between ECDLPs for binary field and prime field curves is due to differences in the required arithmetic operations for each field and the use of search optimization techniques for selected curves.

The Pollard rho method is the best known attack on ECC for curves defined over both binary and prime fields, but the cost of the attack varies based on the properties of the finite field arithmetic. Elliptic curves over prime fields require conventional integer arithmetic operations followed by costly

reduction modulo p. However, the properties of curves over GF(2^m) allow optimizations that can reduce the cost of point addition. Additions and subtractions in the binary finite field are reduced to XOR operations with no carry. Furthermore, the cost of modular reduction is reduced to parallel XOR gates for binary fields. For comparison, reduction modulo p requires sequential additions of carry bits and has a cycle cost approximately equal to that of multiplication. Although attacks on curves over GF(2^m) can be considerably faster than attacks on curves over GF(p), NIST security standards recommend approximately equivalent field sizes for both binary and prime field curves at a given security level [13].

3.3. Parallelization. Van Oorschot and Wiener [5] described a parallelization technique that enables parallel walks on a single curve of Pollard rho algorithm to speed up the computation of ECDLP. The idea is to define a subset of $\langle S\rangle$ as distinguished points (DPs), points which have a distinguishing characteristic. Each parallel walk starts from a distinct random seed point of the form $X_0 = a_0P + b_0Q$ and continues a Pollard rho walk until it encounters a point that satisfies the distinguishing property. Due to properties of the index function of the additive iteration function, a collision between two parallel walks at any point causes the walks to be identical from that point forward. Therefore, checking only DPs for collisions reduces communication and search overhead, while ensuring that any collision between walks is detected. Once a collision is detected, the server must derive the secret key using the collision point and the distinct seed points of the colliding walk. Therefore, whenever a DP is found, it is transmitted to the server along with the random seed point that generated the walk. The parallel Pollard rho method allows distribution of the random walks among multiple processing clients and sharing all DPs found with a central server that performs a collision search. This technique results in a linear speedup as the number of clients increases.

The expected number of DPs required to find a collision is a fraction of the expected path length. This depends on the density of DPs in a point set $\langle S\rangle$, which in turn depends on the chosen distinguishing property. Consider a distinguishing property defined as a point with d-bits fixed in its y-coordinate. This results in a probability of 2^{-d} that a given point is a DP and $|\langle S\rangle|/2^d$ total DPs on the curve. Each walk will require 2^d steps on average to find a DP. Based on the birthday paradox, the number DPs required to find a collision is

$$\phi \geq \frac{\sqrt{(\pi \cdot |\langle S\rangle|)/2}}{2^d}. \tag{7}$$

Since each DP is generated from a walk with a unique random seed point, the number of parallel walks required to solve the ECDLP is also ϕ.

4. Modular Multiplication

We have designed a novel architecture for modular multiplication in a prime field. Typically, hardware solutions use binary field arithmetic, primarily due to the assumption that binary field avoids the costly carry propagation of prime field arithmetic. However, we demonstrate that prime field arithmetic can be efficiently implemented in hardware.

We target the 112-bit secp112r1 elliptic curve in GF($(2^{128} - 3)/76439$), but demonstrate that the proposed architecture can be generalized to perform multiplication modulo any prime of the form $(2^n - m)/k$.

We perform prime field arithmetic using a redundant 128-bit polynomial representation in an affine coordinate system, as proposed by Bernstein et al. in [15] and Bos et al. in [9]. We represent the integers in secp112r1 redundantly in the ring $R = \mathbb{Z}/q\mathbb{Z}$, where $q = p * 76439$. This allows us to perform reduction modulo $2^{128} - 3$, rather than modulo $(2^{128} - 3)/76439$. We perform all arithmetic over the 128-bit field using following method.

Reduction for the unbalanced coefficient $q = 2^{128} - 3$ is a constant multiplication and an addition. Assuming $A = A_1 \cdot 2^{128} + A_0$, then $A \bmod q = A_0 + 3 \cdot A_1$. Integers represented redundantly mod q can be converted into a canonical form in GF(p) by multiplying with $a = 76439$. Let $a \mid q$ and $p = q/a$. Then, $v \bmod p \equiv v \cdot a \bmod q$. Therefore, we start with a unique representation in GF(p), perform arithmetic in R, and canonicalize the results in R to a unique representation by multiplying with $a = 76439$. Similar redundant representation can be used to simplify reduction for any prime $p = (2^n - m)/k$ where $k \neq 0$ by performing arithmetic modulo $q = 2^n - m$ and canonicalizing results by multiplication with k.

4.1. Modular Multiplication Algorithm. Bernstein et al. [15] use noninteger basis for polynomial representation of data to achieve an efficient software implementation. We choose integer basis representation to make the partial product computation uniform across all coefficients, which also makes the design scalable over larger fields. Each l-bit operand is represented as $\sum_{i=0}^{n_a-1} x_i \cdot 2^{i \cdot l_a}$, where $n_a = l/l_a$, l is the length of the operand, and l_a is the integer basis. For secp112r1 curve, $l = 128$ and we discuss optimization of quantity l_a in a subsequent section.

Our approach to modular multiplication is based on the design proposed by Güneysu and Paar in [25]. We use schoolbook multiplication to compute the product $R = F \cdot G$, for $F, G \in$ GF(q). Schoolbook multiplication requires computation of n_a^2 partial products in $O(n^2)$ time, where $R = \sum_{i=0}^{2n_a} 2^{i \cdot n_a} \sum_{j=0}^{i} F_j \cdot G_{i-j}$. We parallelize computation of the partial products to perform n_a multiplications in parallel by multiplying one field of one operand, G, with all fields from the other operand, F. As proposed by Güneysu and Paar in [25], placing G in a shift register and F in a rotating register ensures that each multiplier produces aligned results, that can be directly accumulated. Our design adds an important optimization of the reduction step. We integrate reduction modulo q into the multiplication by multiplying the most significant field of F by m when rotating it to the least significant field. In order to accommodate the multiplication by m of each field of F, we represent each field of F with $l_a + \log_2 m$ bits.

Each partial product is an $l_a + \log_2 m \times l_a$-bit multiplication and produces a $2l_a + \log_2 m$-bit result. After accumulation,

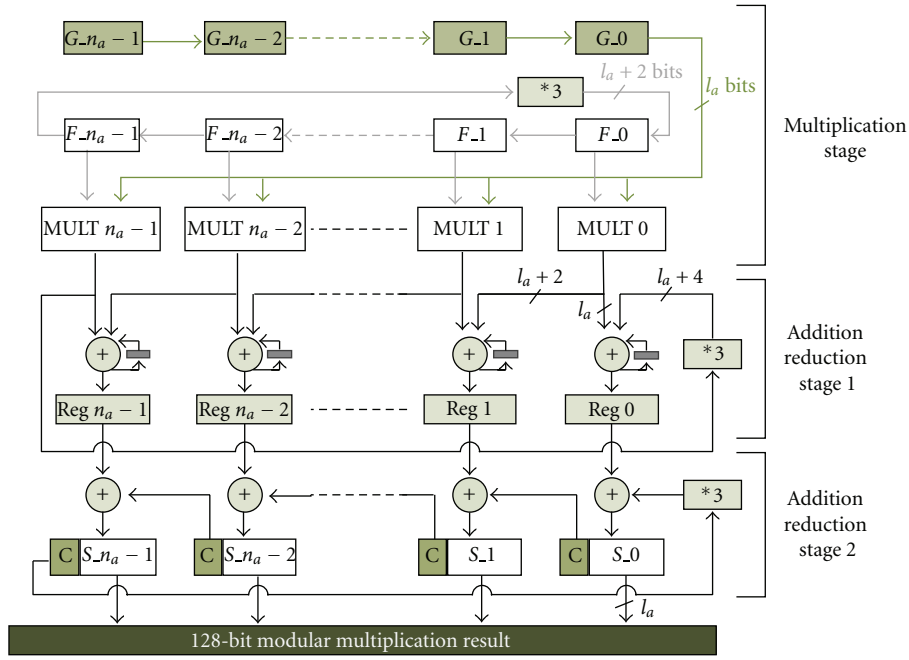

Figure 2: Modular multiplication architecture.

each field of the result can be up to $2l_a + \log_2 m + \log_2 n_a$ bits. We reduce each field to l_a bits using two parallel carry chains. The first carry chain is integrated into partial product accumulation and accumulates the lower l_a-bits of partial product i with the upper bits of partial product $i - 1$. The second carry chain adds the upper bits of each accumulated partial product in column i to the lower bits of the next accumulated partial product in column $i + 1$. In both carry chains, reduction modulo q is achieve by multiplying the upper bits of the most significant field, $i = n_a - 1$, by m before adding them to the least significant field, $i = 0$. The final l-bit result is obtained by concatenating the l_a-bit reduced outputs. The parallel carry chains allow us to overlap the multiplication and reduction stages of the the algorithm to achieve significantly lower latency relative to prior work [25].

4.2. Hardware Architecture. Figure 2 depicts our hardware architecture to implement modular multiplication for secp112r1 elliptic curve with general field size parameters $l = 128$; l_a, n_a. This architecture targets a Xilinx Virtex 5 FPGA platform and makes use of dedicated DSP blocks for high-performance arithmetic. Each DSP block includes a 25×18-bit multiplier and the FPGA fabric includes dedicated routing paths for high-speed connections between DSP blocks [26]. The DSP blocks allow each $l_a + 2 \times l_a$-bit partial product multiplication and accumulation to be computed in a single clock cycle. The modular multiplication takes two 128-bit inputs, F and G, divided into $n_a l_a$-bit fields and produces a 128-bit output of the product reduced modulo $2^{128} - 3$. There are n_a DSP48 multipliers employed to compute partial products of l_a-bit coefficients. Since the n_a^2 partial products are required, it takes n_a multiplication cycles to complete the full unreducedmultiplication result.

Reduction modulo $q = 2^{128} - 3$ adds to the cycle cost of a modular multiplication. By multiplying the shifting operand F_{l_a-1} with 3, we perform the reduction in parallel with the multiplication. For the 128-bit data field with $l_a = 16$ and $n_a = 8$, it takes eight cycles of multiplication and 12 iterations of reduction. This results in a total cycle cost of 20 cycles per modular multiplication. However, our algorithm reduces this cost by overlapping reduction with partial product multiplication and accumulation using the previously described parallel carry adder chains. Thus, the cost has been reduced to 14 cycles, a significant improvement in latency over the architecture proposed by Güneysu and Paar in [25], which takes 70 clock cycles for 256-bit modular multiplication.

4.3. Optimizations. To achieve optimal performance, we evaluate the impact of the integer basis for the polynomial representation on the latency of our design. The performance of the modular multiplier is heavily influenced by the value selected for l_a, which determines the length of each field in the operands and the number of partial products to be computed. Our architecture relies on fast computation of partial products using the available DSP blocks on the FPGA; so performance is also impacted by the suitability of the selected field size to efficiently use these resources.

Modification of our architecture to support different field lengths for polynomial representation is straightforward requiring only that the number of multiplier-adder columns be changed. The number of multiplier-adder columns is given by n_a, which is related to the field length l_a. Increasing the field length requires fewer columns in the architecture and reduces the number of partial products computed. The results of this exploration will be discussed in Section 7.2.

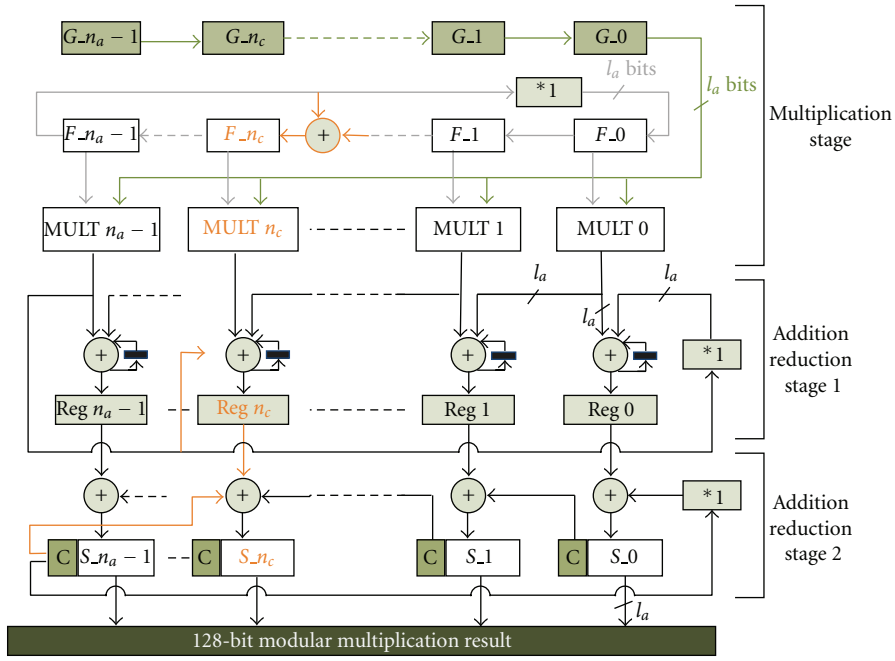

Figure 3: Modular multiplication architecture: extension to NIST P-192 curve.

4.4. Extension to Other Curves. Our modular multiplier is a generalized architecture that can be easily adapted to any overellipticcurve over GF(p), where p has the form $(2^n - m)/k$. We demonstrate this by adapting the architecture for NIST standard P-192 curve, which uses the prime $p = 2^{192} - 2^{64} - 1$. The multiplier architecture is shown in Figure 3. The multiplication for this 192-bit NIST curve is similar to that of secp112r1 curve, but the 192-bit curve needs additional multiplication-adder columns. Additionally, P-192 uses $m = 2^{64} - 1$, rather than $m = 3$ for secp112r1, which requires the following reduction operation. With polynomial field size l_a, we define n_c as the field containing 2^{64}, which is $n_c = 64/l_a$. For $A = A_{n_a-1} \cdot 2^{192} + A_{n_c} \cdot 2^{64} + A_0$, the reduction modulo p is $A \bmod p = (A_{n_c} + An_a - 1) \cdot 2^{64} + (A_0 + A_{n_a-1})$.

We modify the modular multiplier architecture as follows to make it compatible with P-192 curve.

(i) Multiplication factor 3 is replaced by 1 in the rotation path of multiplication stage as well as in the addition reduction feedback path. This reduces the number of bits in each field of operand F to l_a-bits and the length of partial product outputs to $2l_a$-bits.

(ii) In the rotation path, operand field F_{n_a-1} is fed back to two F_{n_c} and F_0. An adder is introduced before F_{n_c} for this purpose. Thus, $F_{n_c} = F_{n_c-1} + F_{n_a-1}$.

(iii) In addition reduction stage 1, carry bits of highest accumulator output (Reg $n_a - 1$) are fed back to both Reg n_c and Reg 0. Similarly in addition reduction stage 2, carry bits of S_{n_a-1} are folded and added back to both S_{n_c} and S_0.

The adapted architecture has a latency of 17 clock cycles at 193 MHz for $l_a = 16$ and has area of 364 Virtex 5 FPGA slices and 12 DSP cores. This is comparable to the multiplier performance for curve secp112r1 (given in Section 7.2) and demonstrates the flexibility of our design to accelerate modular multiplier for general standard prime field elliptic curve.

5. Additional Modular Arithmetic Units for ECDLP

The field operation of a point addition in GF(p) corresponds to a Pollard rho step that consists of four subtractions, one addition, four modular multiplications, and one inversion. This requires design of modular adder/subtractor and modular inversion units. Subsequent sections explain the architecture of these arithmetic modules in detail.

5.1. Addition/Subtraction. We use an integer basis polynomial representation to perform addition and subtraction modulo q using high-speed DSP blocks in the FPGA. We choose a field size of 32, which allows the use of four parallel DSP adder/subtractors to compute the sum of the two 128-bit operands. Our design requires one cycle for the addition/subtraction and one additional cycle for reduction modulo q.

5.2. Inversion. From Fermat's little theorem, it follows that the modular inverse of $z \in \mathbb{F}_q$ can be obtained by computing z^{p-2}. Therefore, computation of the modular inverse requires exponentiation to the $p-2$ power, which is achieved through successive square and multiply operations. For secp112r1 curve, we start with z and compute $z^{((2^{128}-3)/76439)-2}$. We perform exponentiation using a variant of the left to right binary method, also known as square and multiply method

Input: $z \in \mathbb{F}_q$, $e[t:0]$
Output: $z^e \bmod q$
  $r \leftarrow 1$
  for $i = t \rightarrow 0$ **do**
    $r \leftarrow r \cdot r \bmod q$
    if $e[i] = 1$ **then**
      $r \leftarrow r \cdot z \bmod q$
  return r

Algorithm 1: Square and multiply modular exponentiation [24].

[24]. This method scans a bitwise representation of the exponent left to right considering one bit at a time. A squaring is performed for each bit in the exponent, and an additional multiplication by the input operand is performed whenever the current bit is 1. The basic algorithm is shown as Algorithm 1. Due to dependencies between each operation, parallelization of the algorithm is not possible; it needs 112 squarings and 59 multiplications to compute an inversion for secp112r1 curve in 128-bit arithmetic. We improve these figures by applying the following techniques to optimize the inversion operation.

5.2.1. Windowing Optimization. We apply the sliding window method [24] to reduce the number of squarings and multiplications required for inversion. The windowing method is an optimization of the square and multiply algorithm that considers multiple bits of the exponent simultaneously. Computing z^{p-2} with window size k requires precomputation of z^2 and the first $2^{k-1} - 1$ odd powers of z. This method considers up to k bits of the exponent in each iteration and performs a multiplication with one of the precomputed powers of z based on the value of those bits. The sliding window method for modular exponentiation is shown as Algorithm 2.

We achieve optimal performance for a window size of four, which allows inversion with only 108 squarings and 29 modular multiplications. This results in a total of 137 multiplication operations, rather than 171 operations without windowing, a speedup of 1.25.

5.2.2. Vector Inversion. To further reduce the cost of an inversion, we use Montgomery's trick [23], which allows multiple modular inverses to be computed together for significant latency savings. Montgomery's trick is based on the observation that given $(z_1 \cdot z_2 \cdot \cdots \cdot z_M)^{-1}$, the individual inverses $z_1^{-1}, z_2^{-1}, \ldots, z_M^{-1}$ can be easily computed. This allows computation of M modular inverses using $3(M-1)$ multiplications and one inversion. Although the cost of a single inversion is two orders of magnitude higher than multiplication, this approach allows the cost of inversion to be shared across M operations so the marginal cost of an inversion becomes comparable to other arithmetic operations for large vector size M. The algorithm for vector inversion using Montgomery's trick is shown as Algorithm 3.

Input: $z \in \mathbb{F}_q$, $e[t:0]$, $k \geq 1$
Output: $z^e \bmod q$
  // Precomputation
  $p_0 \leftarrow z$, $s \leftarrow z \cdot z \bmod q$
  for $i = 1 \rightarrow 2^{k-1} - 1$ **do**
    $p_i \leftarrow p_{i-1} \cdot s$
  // Inversion
  $r \leftarrow 1$
  $i \leftarrow t$
  while $i \geq 0$ **do**
    if $e_i = 0$ **then**
      $r \leftarrow r \cdot r \bmod q$
      $i \leftarrow i - 1$
    else
      Find longest string $e[i:l]$ where $e[l] = 1$
      and $i - l + 1 \leq k$
      $r \leftarrow r \cdot p_{e[i:l]} \bmod q$
      for $j = 0 \rightarrow (i - l + 1)$ **do**
        $r \leftarrow r \cdot r \bmod q$
      $i \leftarrow i - l - 1$
  return r

Algorithm 2: Sliding window modular exponentiation [24].

Input: $z_1, z_2, \ldots z_M \in \mathbb{F}_q$
Output: $z_1^{-1}, z_2^{-1}, \ldots z_M^{-1} \in \mathbb{F}_q$
  // Preprocessing
  $x_1 \leftarrow z_1$
  for $i = 2 \rightarrow M$ **do**
    $x_i \leftarrow x_{i-1} \cdot z_i \bmod q$
  // Inversion
  $y_M \leftarrow x_M^{-1}$
  // Postprocessing
  for $i = M - 1 \rightarrow 1$ **do**
    $y_i \leftarrow y_{i+1} \cdot z_{i+1} \bmod q$
    $z_{i+1} \leftarrow y_{i+1} \cdot x_i \bmod q$
  $z_1 \leftarrow y_1$
  return $z_1^{-1}, z_2^{-1}, \ldots z_M^{-1}$

Algorithm 3: Vector inversion using Montgomery's trick [23].

We apply this method by implementing a vectorized inversion module that applies Algorithm 3 for any vector size M. To take advantage of vectorized inversion, we design a vectorized point addition datapath that performs random walks on a vector of points simultaneously. We choose vector size $M = 32$, which yields speedup of 19× over 32 individual inversions, and leave optimization of this quantity for future work.

5.3. Squaring. An inversion involves a total of 137 modular multiplications out of which 75% are squaring operations. Having a dedicated squaring unit allows specific optimizations to take advantage of the properties of squaring and reduces the time required to compute an inversion. Consequently, a dedicated module optimized for squaring provides significant acceleration of the point addition operation.

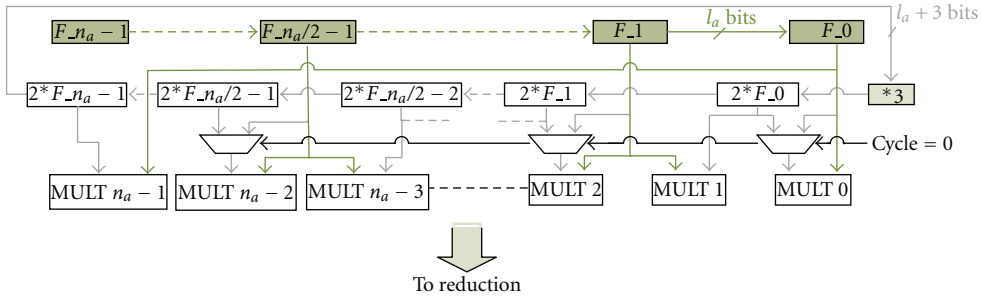

Figure 4: Dedicated squaring architecture.

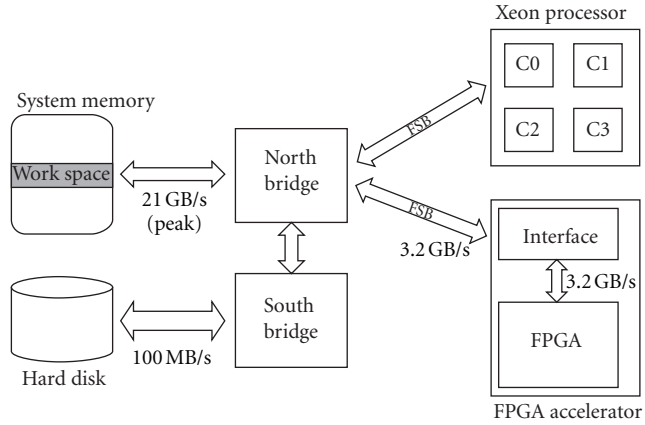

Figure 5: Nallatech system.

Squaring is a special case of the general multiplication of two 128-bit operands F and G. During squaring $F = G$, all off-diagonal partial products, represented as $F_i \cdot F_j$ for $i \neq j$, are computed twice by the conventional schoolbook multiplication method used by our modular multiplier. To reduce computational cost of squaring, we avoid computing any partial product more than once by altering the operand control structure of our multiplier design as shown in Figure 4.

We represent the operand F using the same integer basis for polynomial representation previously described for our modular multiplier. We copy F into a shift register and a rotating register, each with $n_a l_a$-bit fields for polynomial representation, to achieve aligned accumulation of partial products. As shown in Figure 4, the first cycle of multiplication computes the partial products on the diagonal, that is, $F_i * F_j$ where $i = j$. All remaining partial products are off-diagonal. Since off-diagonal partial products are computed twice in schoolbook multiplication, we multiply each field of the rotating register by 2 before computing these partial products. The multiplication by 2 is implemented as a bit shift left at negligible cost. This reduces the number of partial product multiplications from n^2 to $n(n + 1)/2$. The reduction stage for the square module is identical to that of multiplication.

6. ECDLP System Architecture

Our complete ECDLP system is implemented on a Nallatech cointegrated hardware software platform. Figure 5 depicts the architecture of the Nallatech system. It consists of one quad-core Xeon processor E7310 and three Virtex-5 FPGAs (1 xc5vlx110, 2 xc5vsx240t). A fast North Bridge integrates high-speed components, including a Xeon, FPGA, and main memory. A slower South Bridge integrates peripherals into the system, including the hard disk. Both the Xeon and the FPGA can directly access system memory using a Front Side Bus (FSB). The FPGA performs the computationally expensive Pollard rho iterations, while the Xeon processor manages the central database of distinguished points and executes collision search. The communication between software and hardware is carried out only for the exchange of seed points and distinguished points, which minimizes the communication overhead.

6.1. Software Driver. The Xeon processor executes a software driver (in C) and manages software interface to FSB. The software driver mainly handles the communication interface with FPGA; seed point (SP) generation; storage and sorting of DPs. As shown in Figure 5, two-way communication between the Xeon and the FPGA takes place over the FSB.

When the program execution starts, the software calls APIs to configure FPGA card, to initialize the FSB link, and

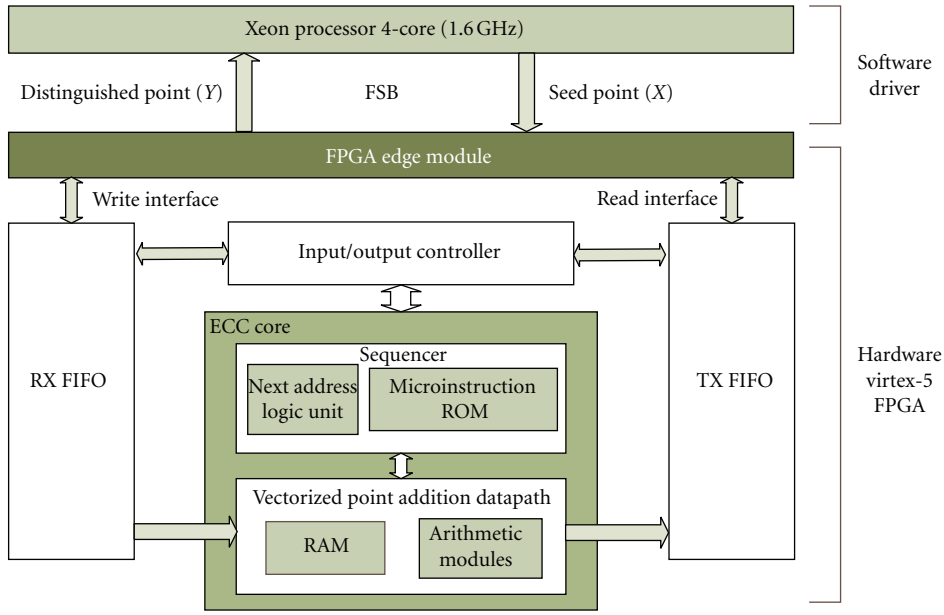

Figure 6: System architecture.

to allocate the workspace memory. It then generates random SPs on the curve E and starts an attack by sending them to the FPGA over FSB. Every point has x- and y-coordinates of 128-bit length each.

The hardware finds a DP for each SP received and sends it to the software along with its corresponding SP. When the software receives SP-DP pair from the FPGA, it performs a collision search among all the received DPs. Once a collision is detected, it computes the secret scalar and reports the solution to the ECDLP. As the software takes care of the central database of DPs, a collision search is conducted in parallel with hardware computations.

6.2. Hardware Accelerator. On the hardware side, as shown in Figure 6, the FPGA edge core provides an interface between the FSB and the ECC core. The edge core consists of control logic and two 256-bit wide FIFOs. The RX FIFO buffers the incoming SPs received over FSB and the TX FIFO stores the DPs found for transmission to the Xeon.

The ECC core performs a random walk by computing the point addition operation iteratively until it finds a DP and stores that in the TX FIFO. We have defined DP as a point with 16 zeros in fixed positions such that $y[123 : 116] = 0$ and $y[59 : 52] = 0$. The probability of a point being distinguished is almost exactly 2^{-16}.

The distinguishing property of points allows to send only few points back to the Xeon, which reduces the communication overhead and minimizes the storage requirement in the hardware. The required bandwidth of communication bus is around 8 K bits/sec, which is well within the range of FSB. In the following are the details of key components in the design.

6.2.1. IO Controller. The IO controller manages the read/write interfaces of TX-RX FIFOs and controls the ECC core operation. It loads the SPs from the RX FIFO to each ECC core and initiates a Pollard rho walk. When a DP is found, the IO controller halts the ECC core operation until a new SP is loaded from the RX FIFO. The computed DPs are buffered in the TX FIFO and then transferred to the Xeon along with corresponding SPs.

6.2.2. ECC Core. The ECC core consists of a microinstruction sequencer and the vectorized point addition (PA) datapath. The core operates on vectors of M-elements to perform M-independent random walks simultaneously. Each walk continues until it finds a DP or crosses the iteration limit, which is currently set to 2^{20}.

6.2.3. Vectorized Datapath. The datapath consists of modular arithmetic operators and memory. Each of these modules is designed to support vectorized point additions. A block diagram of the vectorized datapath is shown in Figure 7. As shown, each of the arithmetic modules interfaces directly to memory in order to load operands and store results. The modular add/sub, modular multiply, and distinguished point check modules are nonvectorized components and operate on inputs from one vector at a time. In order to complete each arithmetic operation over the vectorized walks, computation is serialized and repeated M times for each of these operations. Conversely, the inversion module receives inputs from all M vectors simultaneously and computes the inverse of all inputs together using the previously described vector inversion algorithm.

Based on the sequence of arithmetic operations required for point addition shown in Table 1, we need eight registers ($t1$, $t2$, $t3$, $t4$, Px, Py, Qx, and Qy) to hold the intermediate results for a each point addition. For vector size of M, we use M-entry register files to store these intermediate results.

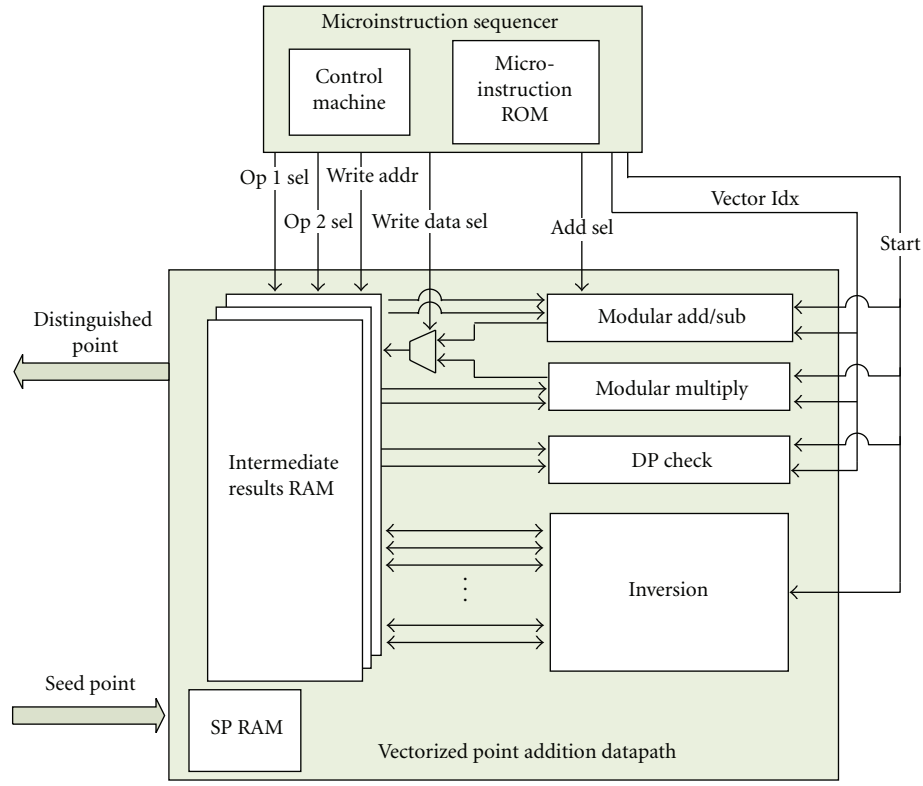

Figure 7: Vectorized point addition datapath with microinstruction sequencer.

Each of the memories is 128 bits wide and has a depth equal to the vector size M. The memory address corresponds to the vector index for each stored data value. We map these register files to the distributed RAMs available in the Xilinx Virtex 5 FPGA. We use an additional 256-bit M-entry memory to store the original seed points for each parallel walk. When a distinguished point is found, that point, along with the original seed point stored in the SP RAM, is send to the IO controller for transmission to the software driver.

6.2.4. Microinstruction Sequencer. We implement a microinstruction sequencer that operates alongside the vectorized point addition datapath to provide control signals necessary to compute vectorized point addition. This allows control of the execution flow of the low-level arithmetic operations for a point addition without modification of the datapath. As shown in Figure 6, the sequencer consists of a microinstruction ROM, which stores instructions to implement point addition, and a next address logic unit (NALU), which determines instruction execution order based on outputs from the datapath. The interconnections between the point addition datapath and microinstruction sequencer are shown in Figure 7.

The vectorized point addition datapath requires a number of control signals to implement each arithmetic operation. A microcoded controller allows us to encode these various control signals into compact form, while maintaining maximum flexibility to modify design parameters including vector size and number of parallel ECC cores. It provides an efficient mechanism to separate our system's control and datapath. Since we use a very specific datapath to perform the particular sequence of operations for point addition, we include only microinstructions needed for the point addition. These correspond directly to the operations shown in Table 1. Common instructions required for general purpose microprocessors, such as *jump, check flag* and are not implemented for our design.

Our microinstruction format is shown in Figure 8. Each microinstruction includes three main segments, arithmetic operation selection, controlled by microinstruction bits 4–0, operand selection, controlled by bits 10–5, and register file write command, controlled by bits 20–11. Decoded control signals from each instruction are connected to datapath inputs as shown in Figure 7.

The next address logic unit (NALU) controls execution order of the microinstructions contained in the ROM. The NALU also handles vectorization of the point addition datapath by repeating each addition, subtraction, multiplication, and distinguished point check operation M times for vector size M. This ensures that the same microinstruction is applied sequentially to each element in the vector. Since the modular arithmetic operations have different latencies, the NALU relies on ready signals generated by datapath components to detect the completion of each instruction. This maximizes flexibility of the microinstruction sequencer and allows the NALU to be independent of the datapath implementation.

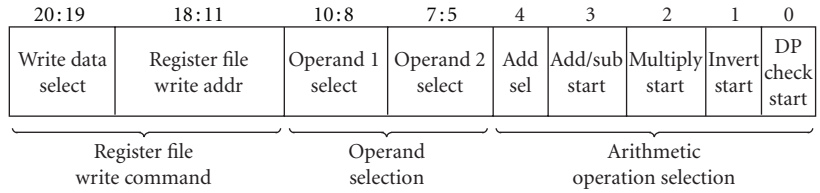

Figure 8: Microcoded instruction format.

In a given cycle, the NALU reads a microinstruction i and issues the corresponding control signals to point addition datapath. These control signals include a start pulse, operand selection, register file write, destination register select, and vector index. Unlike the other control signals, vector index is generated by a finite-state machine within the NALU, rather than decoded directly from microinstruction i. To execute nonvectorized operations, that is all arithmetic operations other than inversion, the vector index is reset and control signals are applied to the datapath. The vector control finite-state machine increments the vector index when the datapath operation completes and asserts the control signals for the next vector element. This continues until the operation is completed for all M vector elements. Then, the vector index resets and the NALU reads the next microinstruction from the ROM.

For an inversion operation, the NALU asserts control signals to load operands and store results from all vector elements. This allows loading of M inputs into the inversion module and writing M results into the corresponding register files. In this way, we efficiently implement vector inversion using Montgomery's trick within the point addition datapath. An inversion is performed only once per M vectors, whereas the other instructions are repeated M times.

6.2.5. Parallel Pollard Rho Walks. We have extended our design to support parallel Pollard rho walks by implementing multiple ECC cores on the FPGA. Each ECC core receives a separate set of SPs and performs a vectorized walk. We have extended the IO controller to support multiple ECC cores so that control signals from all cores are monitored. When a DP is found, the walk by the corresponding ECC core is halted while the IO controller retrieves the DP and loads a new SP. Since each ECC core contains its own microinstruction sequencer module, the multiple cores are not restricted to SIMD execution. When one core is in the process of sending DPs or receiving SPs, the other cores continue execution, avoiding costly latency associated with SIMD configuration.

Our design is scalable to support any number of parallel cores. The IO controller can be configured to support any number of cores. The total number of ECC cores in our design is only limited to the capacity of the FPGA. Our Nallatech computing platform includes two Xilinx Virtex 5 FPGA accelerators and both can be used simultaneously to implement ECC cores to maximize parallelization of Pollard rho walks to solve the ECDLP. Each core performs an independent walk and there is no additional overhead in the point addition iterations due to using multiple cores. Therefore, implementing multiple ECC cores produces a linear increase in performance.

7. Implementation Results

We have implemented our proposed system on the described Nallatech platform in both single- and multicore variants. For demonstration purposes, the seed points that we generate are carefully chosen to be of order 2^{50} [15], which means we would need only 2^{25} steps to solve the ECDLP. This allows us to demonstrate collisions, proving that our solution works.

7.1. Overall Performance. The whole system runs at 100 MHz and uses 5229 slices which is 13% area of the Virtex-5 device xq5vsx240t with a single ECC core. It takes 1.14 microseconds per Pollard rho step and can perform up to 878,000 iterations per second per ECC core.

Our design can easily be extended to include multiple ECC cores performing parallel walks. Each additional ECC core produces as linear increase in performance with negligible control overhead. With 16 ECC cores working in parallel, our system would achieve 14,050,000 iterations per second for secp112r1 ECDLP.

7.2. Evaluation of Modular Multiplier. We have modified the modular multiplier architecture shown in Figure 2 to vary the field length for the polynomial representation, l_a, from 8 bits to 64 bits. Our results are given in Table 2. As shown, increasing the length of the field of the polynomial representation reduces the number of cycles required for the modular multiplication. However, using larger fields also degrades the maximum clock speed of the design. The 64-bit field size requires the fewest cycles per modular multiplication, but achieves worse overall performance due to the low maximum clock frequency.

We also evaluate the impact of the modular multiplier design on overall point addition performance. These results are also shown in Table 2. Modular multiplication is the dominant operation in each point addition, with approximately 85% of iteration cycle count used for multiplication. Thus, we show that decreasing the cycle count of the modular multiplication operation has significant impact on the overall performance of our design. Our results show that best overall performance is achieved for field length of 16-bits when the full system runs at 180 MHz. However, since maximum clock frequency of our complete design is 100 MHz, we select field length of 32 bits, which gives 1.3X increase in performance relative to 16 bit field size.

Güneysu and Paar's architecture described in [25] targets 256-bit prime arithmetic over two fixed NIST primes. Our solution shows an improvement in terms of latency for

Table 2: Modular multiplier performance.

Field length l_a	Computation time (cycles)	Max. clock frequency (MHz)	Area (slices, DSPs)	Latency per iteration (μs)	System performance (PA/s)	ECC core area (slices, DSPs)
8	21	181	263, 18	1.24 (237 cycles)	808 K	5199, 114
16	14	181	217, 10	0.790 (151 cycles)	1.27 M	4862, 66
32	7	104	253, 20	1.09 (114 cycles)	913 K	5229, 130
64	3	68	370, 36	1.17 (80 cycles)	854 K	5776, 214

Table 3: Comparison with software ECDLP implementations.

Platform	Latency per iteration (ns)	Performance (PA/s)
Cell processor at 3.192 GHz, secp112r1 curve [15]	113 (362 cycles)	8.81 M
Cell processor, at 3.192 GHz, secp112r1 curve [9]	142 (453 cycles)	7.04 M
Cell processor, at 3.192 GHz, ECC2K-130 binary field [10]	233 (745 cycles)	4.28 M
Our system, secp112r1	1140 (114 cycles)	878 K: single core 14.05 M: 16 cores

Table 4: Comparison with Hardware ECDLP implementations (per core).

Platform	Target curve	Performance (PA/s)	Area (Slices)	Area (BRAMs)	Area (DSPs)
Spartan-3 [16]	Binary (130 bit)	111 M	26,731	20	0
Spartan-3 [20]	Binary (113 bit)	20 M	13,900	18	0
Spartan-3 [18]	Prime (160 bit)	46.80 K	3,230	15	0
Spartan-3 [18]	Prime (128 bit)	57.80 K	2,520	16	0
Spartan-3 [19]	Prime (160 bit)	50.12 K	2,660	Not given	0
Virtex-5, our system	Prime (112 bit)	878 K	5,229	9	130

the important ECC arithmetic operations. Assuming the cycle cost for 256-bit arithmetic as twice of that for 128-bit arithmetic (worst-case scenario), our architecture has cycle cost of 14 for a modular multiplication and 228 for the point addition. This is 5X and 3.5X times lower latency for a modular multiplication and point addition operation respectively, than those of the design in [25].

7.3. Comparison with Prior Work. Due to inherent differences between elliptic curves defined over binary and prime fields, performance comparison among various ECDLP implementations is not straightforward. Performance figures are also influenced by a variety of other factors, including target platform, curve size, and coordinate system. To minimize the impact of the target platform and coordinate system, we compare performance of different ECDLP implementations based on the total number of point addition operations performed per second.

Table 3 compares our solution with other software implementations. It shows that a multicore implementation of our design using 16 ECC cores achieves 1.6X improvement over the fastest existing software attack on secp112r1 curve.

Comparison with prior hardware ECDLP implementations is shown in Table 4. Our design achieves high-performance relative to other prime field solutions. When adjusting for curve size, we demonstrate significant speedup over prior designs by [18, 19].

We also show comparison with existing binary field solutions. However, it is difficult to make a fair comparison due to significant differences between binary and prime field arithmetic. In particular, properties of binary field arithmetic and reduction are well-suited for hardware designs, which allows them to achieve much higher performance.

The solution proposed in [16] reports 111 M point addition iterations per second. The authors of [16] claim their system is capable of solving ECC2K-130 within a year using five COPACOBANA machines. Similarly, the solution reported in [20] achieves 20 M iterations per second. We assume that the difference of performance figures exists due to factors including binary field arithmetic, different curve sizes, and use of pipelined architectures.

8. Conclusion

We successfully demonstrate a complete multicore ECC cryptanalytic machine to solve ECDLP on a hardware-software cointegrated platform. We also implement a novel architecture on hardware to perform modular multiplication over prime field and this is the most efficient implementation reported at present for prime field multiplication. We then

present a generalized modular multiplication architecture for primes of the form $(2^n - m)/k$ and demonstrate its application to NIST standard P-192 curve. This work also demonstrates the use of microinstruction-based sequencing logic to support a vectorized point addition datapath with variable vector sizes. We compare our performance results with the previous implementations and show that a multicore implementation of our solution has competitive performance relative to existing hardware and software solutions.

Acknowledgments

An earlier version of this paper appeared as "*An Integrated prime-field ECDLP hardware accelerator with high-performance modular arithmetic units*" in the Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs (RECONFIG) [27]. This research was supported in part by the National Science Foundation Grant no. 0644070.

References

[1] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology CRYPTO 85 Proceedings*, H. Williams, Ed., vol. 218 of *Lecture Notes in Computer Science*, pp. 417–426, Springer, Berlin, Germany, 1986.

[2] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[3] I. Blake, G. Seroussi, N. Smart, and J. W. S. Cassels, *Advances in Elliptic Curve Cryptography*, London Mathematical Society Lecture Note Series, Cambridge University Press, New York, NY, USA, 2005.

[4] J. M. Pollard, "Monte carlo methods of index computation (mod p)," *Mathematics of Computation*, vol. 32, no. 143, pp. 918–924, 1978.

[5] P. C. Van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications," *Journal of Cryptology*, vol. 12, no. 1, pp. 1–28, 1999.

[6] E. Teske, "On random walks for Pollard's rho method," *Mathematics of Computation*, vol. 70, no. 234, pp. 809–825, 2001.

[7] E. Teske, "Speeding up Pollard's rho method for computing discrete logarithms," in *Algorithmic Number Theory*, J. Buhler, Ed., vol. 1423 of *Lecture Notes in Computer Science*, pp. 541–554, Springer, Berlin, Germany, 1998.

[8] R. P. Brent, "An improved Monte Carlo factorization algorithm," *BIT Numerical Mathematics*, vol. 20, pp. 176–184, 1980.

[9] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery, "On the security of 1024-bit RSA and 160-bit elliptic curve cryptography," Report 2009/389, IACR Cryptology ePrint Archive, 2009, http://eprint.iacr.org/2009/389.

[10] J. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe, "ECC2K-130 on cell CPUs," in *Progress in Cryptology—AFRICACRYPT 2010*, D. Bernstein and T. Lange, Eds., vol. 6055 of *Lecture Notes in Computer Science*, pp. 225–242, Springer, Berlin, Germany, 2010.

[11] D. V. Bailey, L. Batina, D. J. Bernstein et al., "Breaking ECC2K-130," Report 2009/541, IACR Cryptology ePrint Archive, 2009, http://eprint.iacr.org/2009/541.

[12] D. Bernstein, H.-C. Chen, C.-M. Cheng et al., "ECC2K-130 on Nvidia GPUs," in *Progress in Cryptology—INDOCRYPT 2010*, G. Gong and K. Gupta, Eds., vol. 6498 of *Lecture Notes in Computer Science*, pp. 328–346, Springer, Berlin, Germany, 2010.

[13] *FIPS 186-3: Digital Signature Standard (DSS)*, National Institute of Standards and Technology (NIST), June 2009, http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf.

[14] *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography Group (SECG), January 2010, http://www.secg.org/download/aid-784/sec2-v2.pdf.

[15] D. Bernstein, T. Lange, and P. Schwabe, "On the correct use of the negation map in the Pollard rho method," in *Public Key Cryptography—PKC 2011*, D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, Eds., vol. 6571 of *Lecture Notes in Computer Science*, pp. 128–146, Springer, Berlin, Germany, 2011.

[16] J. Fan, D. V. Bailey, L. Batina, T. Güneysu, C. Paar, and I. Verbauwhede, "Breaking elliptic curve cryptosystems using reconfigurable hardware," in *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 133–138, IEEE Computer Society, September 2010.

[17] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA a cost-optimized parallel code breaker," in *Cryptographic Hardware and Embedded Systems—CHES 2006*, L. Goubin and M. Matsui, Eds., vol. 4249 of *Lecture Notes in Computer Science*, pp. 101–118, Springer, Berlin, Germany, 2006.

[18] T. Gueneysu, C. Paar, and J. Pelzl, "Attacking elliptic curve cryptosystems with special-purpose hardware," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '07)*, pp. 207–215, ACM, New York, NY, USA, February 2007.

[19] T. Güneysu, C. Paar, and J. Pelzl, "Special-purpose hardware for solving the elliptic curve discrete logarithm problem," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 2, pp. 8:1–8:21, 2008.

[20] G. Meurice de Dormale, P. Bulens, and J.-J. Quisquater, "Collision search for elliptic curve discrete logarithm over GF(2^m) with FPGA," in *Cryptographic Hardware and Embedded Systems—CHES 2007*, P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, pp. 378–393, Springer, Berlin, Germany, 2007.

[21] P. Majkowski, T. Wojciechowski, M. Wojtyński, M. Rawski, and Z. Kotulski, "Heterogenic distributed system for cryptanalysis of elliptic curve based cryptosystems," in *Proceedings of the 19th International Conference on Systems Engineering (ICSEng '08)*, pp. 300–305, August 2008.

[22] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery, "Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction," *International Journal of Applied Cryptography*, vol. 2, no. 3, pp. 212–228, 2012.

[23] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[24] D. R. Hankerson, S. A. Vanstone, and A. J. Menezes, *Guide to Elliptic Curve Cryptography*, Springer, New York, NY, USA, 2004.

[25] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Cryptographic Hardware and Embedded Systems—CHES 2008*, E. Oswald and P.

Rohatgi, Eds., vol. 5154 of *Lecture Notes in Computer Science*, pp. 62–78, Springer, Berlin, Germany, 2008.

[26] *Virtex-5 FPGA XtremeDSP Design Considerations*, Xilinx, Inc., January 2010, http://www.xilinx.com/support/documentation/user guides/ug193.pdf.

[27] S. Mane, L. Judge, and P. Schaumont, "An integrated prime-field ECDLP hardware accelerator with high-performance modular arithmetic units," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, P. M. Athanas, J. Becker, and R. Cumplido, Eds., pp. 198–203, IEEE Computer Society, December 2011.