

An Integrated Prime-field ECDLP Hardware Accelerator with High-performance Modular Arithmetic Units

Suvarna Mane
ECE Department
Virginia Tech
Blacksburg, USA
suvarnam@vt.edu

Lyndon Judge
ECE Department
Virginia Tech
Blacksburg, USA
lvjudge1@vt.edu

Patrick Schaumont
ECE Department
Virginia Tech
Blacksburg, USA
schaum@vt.edu

Abstract—This paper reports a successful demonstration of Pollard rho algorithm on a hardware-software co-integrated platform. It targets the Elliptic curve discrete logarithmic problem (ECDLP) for a NIST-standardized curve over 112-bit prime field. To the best of our knowledge, this is the first report on fully functional, demonstrated hardware-accelerated ECC cryptanalytic engine. Our implementation uses a highly optimized software implementation as reference [1] and develops a hardware version of it. This paper also describes a novel, generalized architecture for polynomial-basis multiplication over prime field and its extension to a dedicated square module. The resulting modular multiplier completes the multiplication within 14 clock cycles, which is 2.5X lower latency over earlier work [2]. We demonstrate our design on a Nallatech FSB-Compute platform with Virtex-5 FPGA. The implementation efficiently utilizes the dedicated DSP48 cores available in the used FPGA device. The measured performance of the resulting design is 151 cycles per Pollard-rho step at 100MHz and upto 660K iterations per second per ECC core. With a multi-core implementation of our design, the performance can be comparable with that of the software implementation on a Cell processor [1]. Though the primary target of this implementation is 112-bit prime field, its design strategy can be applied to other prime field moduli.

Index Terms—FPGA; Elliptic curve discrete logarithmic algorithm (ECDLP); Pollard rho; Prime field arithmetic; Hardware software co-design;

I. INTRODUCTION

Elliptic curve cryptosystems (ECC), independently introduced by Miller [11] and Koblitz [9], have now found significant place in the academic literature and practical applications. Their popularity is mainly because of their shorter key-sizes, which offer the same level of security as other conventional cryptosystems such as RSA. The security of ECC relies on the difficulty of Elliptic Curve Discrete Logarithmic Problem (ECDLP) [10]. By definition, ECDLP is to find an integer n for two points P and Q on an elliptic curve E such that

$$Q = [n]P \quad (1)$$

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

The Pollard Rho method [12], [13] is the strongest known attack against ECC today. This method solves ECDLP by generating points on the curve iteratively, any of which have

the property $X = [a]P + [b]Q$. When the same point is encountered twice, for different $[a]$ and $[b]$, the ECDLP is solved.

There have been different approaches to implement Pollard rho algorithm on the software and hardware platforms. Most of the solutions are implemented on software platforms using general purpose workstations, such as clusters of PlayStation3 [4], Cell CPUs [3], GPUs [8]. 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 [17]. 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 $(2^k - q)/m$.

Our work also 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, 1.6GHz) with a tightly coupled Virtex-5 (xq5vsvx240t) FPGA. The hardware runs at 100MHz and utilizes 4773 slices per ECC core.

II. RELATED WORK

The software solution proposed by Bernstein on CELL platform is the fastest software solution at present, to solve the ECDLP over secp112r1 curve [1]. It uses the negation map and non-integer polynomial-basis arithmetic to report the speedup over a similar solution by Bos [4]. Both of these

software solutions use prime field arithmetic in an affine co-ordinate system, and they exploit the SIMD architecture and rich instruction set. Another software solution by Bos [3] describes the implementation of parallel Pollard rho algorithm on Synergistic Processor Units of Cell Broadband Engine Architecture to approach the ECC2K-130 certicom challenge.

Among hardware-platform-based solutions, Bulens et al. proposes an FPGA solution to attack the ECC certicom challenge for $\text{GF}(2^{79})$ [5]. Though it discusses the hardware-software integration aspect of the solution, the authors did not confirm if their system was operational. Fan proposes the use of a normal-basis, binary field implementation to solve ECC2K-130 [6].

Another binary field solution, for the COPACOBANA platform, targets the 160-bit curve [14]. Since a curve of this size would require a single COPACOBANA platform to run for 7.62×10^9 years, the authors did not demonstrate a collision that can validate their design. Guneyusu et al. propose an architecture to solve ECDLP over prime fields using FPGAs and analyze its estimated performance for different ECC curves [7]. A three-layer hybrid distributed system is described by Piotr et al. to solve ECDLP over binary field [17]. 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 briefly discuss the background of parallel Pollard-rho algorithm and then we present the details of some of the modular arithmetic modules in section IV. In section V, we discuss the hardware-software integrated system architecture with its key components. Section VI shows implementation results, including measured performance for secp112r1 ECDLP and we conclude the paper in section VII.

III. POLLARD RHO ALGORITHM

Let E be an elliptic curve over \mathbb{F}_q , $X \in E(\mathbb{F}_q)$ a point of order l and $Q = [n]P$, where $P, Q \in \langle X \rangle$. Here, q is a prime and $l, n \in \mathbb{Z}$. ECDLP is defined as: to find n when P and Q are known.

The Pollard rho algorithm [12] uses a pseudo-random iteration function $f: \langle X \rangle \rightarrow \langle X \rangle$ and calculates a finite number of points on the curve. It starts a walk from a random point (seed point) on the curve of the form $R_0 = [a_0]P + [b_0]Q$, where a_0 and b_0 are generated from a random seed s . It then iteratively computes $R_{i+1} = f(R_i)$ until it encounters a point on the curve twice: eventually, walk ends in a cycle. The name of the algorithm, rho, expresses the Greek letter ρ , which shows a walk ending in a cycle.

The collision point is located where the cycle starts. Therefore the underlying idea of this algorithm is to search for two distinct points on the curve such that

$$[a_i]P + [b_i]Q = [a_j]P + [b_j]Q.$$

The iteration function is constructed in such a way that a_d and b_d can be computed using a_0 and b_0 . A Pollard-rho step corresponds to an iteration function and is often defined as a

point addition i.e. $R_{i+1} = R_i + r$, where r is a precomputed, linear combination of P and Q , for example, $[c_i]P + [d_i]Q$.

When a collision occurs, two different linear combinations of R_d are computed using a_d and b_d of the collided points. The solution then can be obtained as

$$n = \left[\frac{a_{d_1} - a_{d_2}}{b_{d_1} - b_{d_2}} \right] \text{mod } l \quad (2)$$

Due to the birthday-paradox, the expected length of a walk before a collision is found, is proportional to $\sqrt{|\langle X \rangle|}$.

A. Parallelization

Van Oorschot [15] 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 $\langle X \rangle$ as distinguished points (DPs), points which have a distinguishing characteristic. For example, a DP could be defined as a point with a given number of leading zero-bits in its x-coordinate. This method allows to distribute the random walks among multiple processing clients and share the DPs found by them with a central server, which then performs a collision search. This technique results in a linear speed up as the number of clients increases.

Multiple random walks are continued till two different seed points reach the same distinguished point R_d . 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 $\langle X \rangle$, which in turn depends on the chosen distinguishing property.

IV. ECC ARITHMETIC MODULE ARCHITECTURE

We target prime field arithmetic using polynomial representation in an affine co-ordinate system. Typically, hardware solutions use binary field arithmetic, primarily because of the assumption that binary field avoids costly carry propagation. Our representation for prime-field arithmetic is based on [1], which has similar properties.

Although, the target curve considered in this paper is secp112r1 over 112-bit in $\text{GF}((2^{128} - 3)/76439)$, all the arithmetic operations are performed over 128-bit field. After each iteration, the result is mapped to 112-bit to check it against the distinguishing property. This 128-bit to 112-bit conversion is obtained by the canonicalization i.e. multiplying a result with 76439 [1].

Table I
ECC ARITHMETIC OPERATIONS

Operation	Cycle cost
Addition/Subtraction	2
Modular Multiplication	14
Square	11
Inversion	1594

The point addition operation corresponds to a Pollard-rho step that consists of four subtractions, one addition, four modular multiplications, and one inversion. Table I lists these

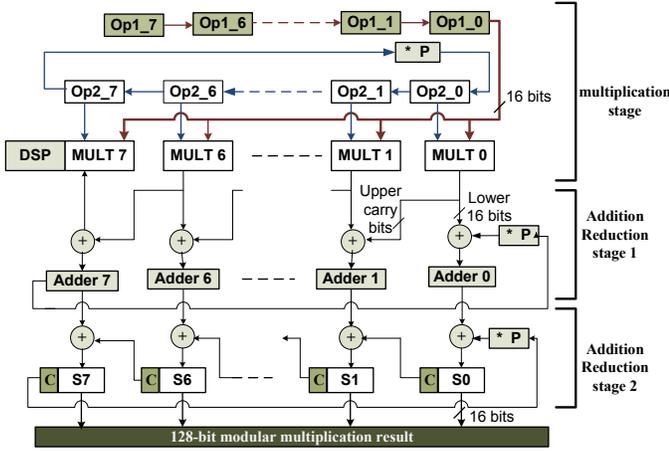


Figure 1. Modular Multiplication Architecture

arithmetic operations along with their cycle costs to compute a single operation. Subsequent sections explain the architecture of arithmetic modules in detail.

A. Modular Multiplication

Bernstein [1] uses the non-integer basis for polynomial representation of data to achieve an efficient software implementation. We choose 16-bit coefficient representation to make the partial product computation uniform across all the coefficients, which also makes the design scalable over larger fields. These multipliers are mapped on the dedicated DSP48 cores available in the used Virtex-5 devices for efficient implementation. The 128-bit data is represented as

$$X = \sum_{i=0}^{n_A-1} x_i \cdot 2^{i \cdot l_A} \text{ where, } n_A = l/l_A, \quad (3)$$

$$l = 128 \text{ and } l_A = 16.$$

Figure 1 depicts the architecture of the modular multiplication. It takes two 128-bit inputs ($Op1$ and $Op2$) broken into 16-bit coefficients and gives 128-bit reduced output. There are eight DSP48 multipliers employed to find partial products of 16-bit coefficients. As it needs to compute n_A^2 (64 in our case) partial products to get multiplication result, it takes eight multiplication cycles to get unreduced multiplication result.

A similar architecture was presented earlier by Guneyusu [2]. However, Figure 1 presents an important optimization of the reduction step. Since we are computing $Op1 * Op2 \bmod (2^{128} - p)$, the reduction adds to the cycle cost of a modular multiplication ($p = 3$ in our case). By multiplying the shifting operand $op2_7$ with 3, we perform the reduction in parallel with the multiplication.

For the 128-bit data field with eight DSP multipliers, it takes eight cycles of multiplications and 12 iterations of reduction. This needs a cycle cost of 20 per modular multiplication. As depicted in Figure 1, this cost has been reduced to 14 cycles, by overlapping reduction with partial multiplications. This is a significant improvement in terms of latency over

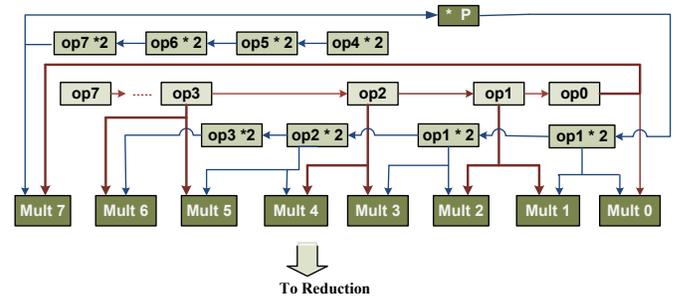


Figure 2. Dedicated Square Architecture

Guneyusu's architecture [2], which takes 70 clock cycles for 256-bit modular multiplication.

The reduction has been achieved by an adder chain, which adds lower 16 bits of i th partial product with the upper carry bits of $(i-1)$ th one. The carry bits of the highest coefficient $Adder7$ are multiplied by 3 before adding them to the lowest coefficient $Adder0$. The final 128-bit result can be obtained by concatenating these eight 16-bit reduced outputs. A two stage pipeline of adder chains is employed to reduce the critical path of the reduction stage.

This architecture supports the generalized modular multiplication for any p over 128-bit prime field. It can easily be extended over larger prime fields by adding additional multiplier-adder columns, in which case, the performance might differ with the number of multiplier-adder columns.

B. Dedicated Square Unit

Table I shows that, an inversion is the most expensive operation in a Pollard-rho step. Therefore, optimizing this operation results in an improvement of the overall system performance. An inversion involves a total of 137 modular multiplications, out of which, 75% are the squaring operations. Having a dedicated optimized squaring unit reduces the effective time of inversion and consequently accelerates the point addition operation.

The squaring operation needs only half as many partial products as multiplication, so we modified the multiplication architecture to get an optimized square module as shown in Figure 2. It involves only 5 multiplication iterations instead of 8. The reduction stage for the square module is similar to that of multiplication. With this architecture, a square operation is completed in only 11 cycles, which achieves the speed-up of 1.2 over the multiplication architecture. Since the majority of operations required for inversion are squares, this translates to a significant reduction in the cycles cost of an inversion.

C. Vectorized Inversion

From the Fermat's little theorem, it follows that the modular inverse of $P \in E(\mathbb{F}_q)$ can be obtained by computing $P^{(q-2)}$. It needs 112 squarings and 59 multiplications to find an inversion for secp112r1 curve in 128-bit arithmetic. We used following techniques to optimize the inversion operation.

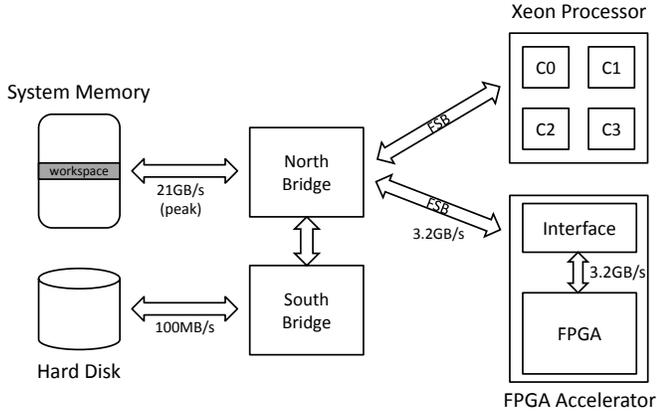


Figure 3. Nallatech System

1) *Windowing Optimization*: A windowing method allows to reduce the number of squarings and multiplications needed to invert an input. For a window-size of four, we achieve an inversion operation in 108 squarings and 29 modular multiplications. It takes 1594 clock cycles for an inversion as opposed to 2058 cycles without windowing, providing a speed-up of 1.3.

2) *Montgomery trick*: To further reduce the cost of an inversion, we use Montgomery’s trick [16], which enables to trade M inversions for $3(M - 1)$ multiplications and one inversion. It allows to vectorize multiple random walks together and run them simultaneously on a single ECC core, to share the inversion cost.

With the large vector size, the inversion cost per iteration becomes small compared to other operations such as multiplication. We select vector size of 32 and optimization of this quantity is a future work.

V. SYSTEM ARCHITECTURE

The system is implemented on a Nallatech computing platform. The FPGA performs the computationally expensive Pollard-rho iterations, whereas the host processor manages the central database and executes collision search. The communication between software and hardware is carried out only for the exchange of seed points and distinguished points, which reduces the communication overhead.

A. Nallatech Platform

Figure 3 depicts the architecture 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).

A Field Programmable Gate array (FPGA) is used for computing additive random walks on elliptic curve, while a host Xeon processor executes the software driver.

B. Software Implementation

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

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

C. Hardware Implementation

On the hardware side, as shown in Figure 4, FPGA edge core provides an interface between the FSB and the ECC core. It consists of a control logic and two 256-bit wide FIFOs. RX FIFO buffers the incoming SPs and TX FIFO stores the DPs found, to send them back 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, which has y -coordinate with 16 zeros. 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 8Kbits/sec, which is well within the range of FSB. Following are the details of key components in the design.

1) *IO controller*: The IO controller manages the read/write interfaces of TX-RX FIFOs and controls the ECC core operation. It feeds the SPs from RX FIFO to the 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.

2) *ECC Core*: The ECC core consists of a micro-instruction sequencer and the point-addition (PA) datapath. It performs a random walk by computing point additions iteratively until it finds a DP or crosses the iteration limit (which is currently set to 2^{20}).

3) *Sequencer*: This secondary controller executes the micro-instruction sequence stored on a ROM and accordingly issues the control signals to the PA datapath module. This way, it controls the execution flow of the low-level arithmetic operations for a point addition. The micro-coded architecture

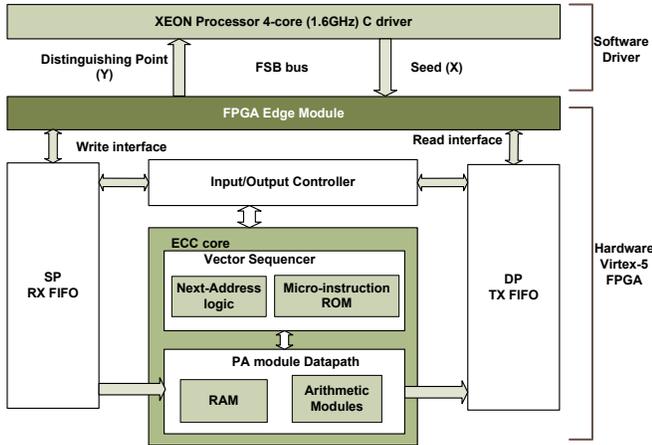


Figure 4. System Architecture

adds to the flexibility of the micro-instruction sequence to support different vector sizes (N). It also supports scalability of the design by providing a mechanism to control multiple ECC cores in a SIMD (Single Instruction Multiple Data) fashion. The vector size is a generic parameter of the design.

Table II summarizes the micro-instruction sequence for a single point addition. Here, $t1$, $t2$, $t3$, $t4$, Px , Py , Qx and Qy represent N -entry register files. The register files Px , Py , Qx and Qy hold the x - and y - coordinates of the points to be added and remaining register files store the intermediate results.

The next-address logic (NAL) controls the execution of microinstructions from a ROM. Every micro-instruction consists of N phases, corresponding to the N entries of a vector. The same micro-operation is applied to every element of the vector. Furthermore, every micro-operation can have a variable execution time. This allows the micro-instruction concept to be applied to all the operations required for point addition regardless of latency.

The NAL module reads a micro-instruction i and issues the corresponding control signals to PA datapath N times. These control signals include a start pulse, an operand select, the opcode of a micro-operation, a write pulse to store the results, a destination-register select and a vector phase n_i , to which the operation belongs.

For an inversion operation, NAL module extends start and write signals for N clock cycles, each with different vector phase n_i . This enables to load N inputs into inversion module and write N results into the corresponding register files. An inversion is performed only once per N vectors, whereas the other instructions are executed N times.

As the datapath of PA iteration is fixed, there is no need of the conventional conditional micro-instructions such as *jump*, *check flag* etc. This makes the instruction-set simpler.

4) *Datapath*: The datapath consists of modular arithmetic operators and memory. We carefully designed each of these sub-blocks to support vectorized point additions. A vectorized point addition allows the execution of multiple random walks

Table II
MICRO-INSTRUCTION SEQUENCE FOR POINT ADDITION: $P + Q$

Instruction	Function performed
canonicalization	$Py * 76439$
Subtraction	$t1 = Py - Qy$
Subtraction	$t2 = Px - Qx$
Inversion	$t2 = \text{invert}(t2)y$
Addition	$t3 = Px + Qx$
Modular Multiplication	$t4 = t2 * t1$
Modular Square	$t1 = t4 * t4$
Subtraction	$t1 = t1 - t3 : Px (i+1)$
Subtraction	$t2 = Px - t1$
Modular Multiplication	$t3 = t2 * t4$
Subtraction	$t3 = t3 - Py : Py (i+1)$

simultaneously on a single ECC core, with only one inversion per N point-additions.

As shown in Table II, we need eight registers ($t1$, $t2$, $t3$, $t4$, Px , Py , Qx , Qy) to hold the intermediate results for a single point addition. For vector size of N , we use N -entry register files. For efficient implementation, these N -entry register files are mapped to the distributed RAMs available in FPGA. Each of these memories is 128-bit wide and has a depth equal to the vector size N . The address of an entry in the memory corresponds to the vector number, to which the content belongs.

VI. IMPLEMENTATION RESULTS

Though our work is dedicated to prime $p = (2^{128} - 3)/76439$, the same solution can work with little modifications for any curve of the form $y^2 = x^3 - 3x + b$. For the demonstration purpose, the seed points that we generate are carefully chosen to be of order 2^{50} [1], which means we would need only 2^{25} steps to solve the ECDLP. This allows us to demonstrate collisions, proving that our solution works.

A. Overall Performance

The whole system runs at 100 MHz and utilizes 4773 slices which is 12.7% area of the Virtex-5 device xq5vsvx240t with a single ECC core. It takes 1.5 microseconds per Pollard-rho step and can perform upto 660K iterations per second per ECC core. With 16 ECC cores working in parallel, our system would need 176 years to solve secp112r1 ECDLP.

The Guney's architecture described in [2] targets 256-bit prime arithmetic over two fixed NIST primes. Our solution shows an improvement over it in terms of latency for the important ECC arithmetic operations. Assuming the cycle cost for 256-bit arithmetic as twice of that for 128-bit arithmetic (worst case scenario), we can see that our architecture has cycle cost of 28 for a modular multiplication and 302 for the point addition. This is 2.5X and 3X times lower latency for a modular multiplication and point addition operation respectively, than those of the design in [2].

The performance comparison among various implementations is not a straightforward process, as different solutions

Table III
COMPARISON WITH SOFTWARE IMPLEMENTATIONS

Platform	Time/PA in ns	Iterations/sec
Cell processor @3.192GHz, secp112r1 curve [1]	113 (362 cycles)	8.81M
Cell processor, @3.192GHz, secp112r1 curve [4]	142 (453 cycles)	7.04M
Cell processor, @3.192GHz, ECC2K-130 Binary Field [3]	233 (745 cycles)	4.28M
Our system, Secp112r1	1500	660K: single core 10.56M: 16 cores

target different curves and different co-ordinate systems. Also the performance figures are specific to the target platform, the size of the target curves and an underlying arithmetic (i.e. binary or prime field).

B. Comparison with Previous Software Implementations

Table III compares our solution with other software implementations. It shows that our results can be comparable with those of software if we have multiple ECC cores in parallel. The listed multi-core performance is an estimate; our measured results are for a single ECC core only.

C. Comparison with Hardware Implementation

As shown in Table IV, the solution reported in [6] claims to have 111M iterations per second. It also claims to solve ECC2K-130 within a year with five COPACOBANA machines, but the system demonstration is not reported.

Similarly, the solution reported in [5] claims to have 100M iterations per second based on paper design. We assume the difference of performance figures exists due to the factors, such as, binary field arithmetic, different curve sizes and use of pipelined architectures. We can see that ours is the only solution at present, which demonstrates the Pollard rho algorithm successfully on a hardware-software integrated platform.

VII. CONCLUSION

We successfully demonstrate a complete ECC cryptanalytic machine to solve ECDLP on hardware-software co-integrated 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. This architecture is further extended to a dedicated square module. This work also demonstrates the use of micro-instruction based sequencing logic to support different vector sizes and to control multiple ECC cores in SIMD fashion. We compare our performance results with the previous hardware implementations and show that our solution can have a comparable performance with multi-core implementation. The proposed system can further be used to demonstrate the prime arithmetic over other curves of different sizes.

VIII. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation Grant no. 477634

Table IV
COMPARISON WITH HARDWARE IMPLEMENTATIONS (PER ECC CORE)

Platform	Arithmetic	Iterations /sec	Area (slices)	Demonstrated
Spartan-3 [14]	Prime (160-bit)	47.28K	3034	Unclear
Spartan-3 [7]	Prime (160-bit)	50.12K	2660	Unclear
Spartan-3 [6]	Binary (130-bit)	claims 111M	26731	Paper design
Virtex-4 [5]	Binary (79-bit)	claims 100M	22236 Slices 30 BRAMs	Paper design
Virtex-5 our system	Prime (112-bit)	660K	4773 Slices 9 BRAMs 62 DSP48	Demonstrated

REFERENCES

- [1] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the correct use of the negation map in the Pollard Rho method. *IACR ePrint (2011)*, <http://eprint.iacr.org/2011/003.pdf>.
- [2] Tim Guneysu, Christof Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. *CHES 2008*, LNCS 5154, pp. 62-78, 2008.
- [3] Bos Joppe W., Kleinjung Thorsten, Niederhagen Ruben, Schwabe Peter. ECC2K-130 on Cell CPUs. (*2010 Jan 1*) vol. 6055, pp. 225-242, 2010.
- [4] Bos, J.W., Kaihara, M.E., Montgomery, P.L. Pollard rho on the PlayStation 3. In: Workshop Record of *SHARCS 2009*, pp. 35-50 2009. <http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf>.
- [5] Philippe Bulens, Gueric Meurice de Dormale, Jean-Jacques Quisquater. Hardware for Collision Search on Elliptic Curve over GF(2m). *SHARCS'06* April, 2006.
- [6] Junfeng Fan, Daniel V. Bailey, Lejla Batina, Tim Guneysu, Christof Paar and Ingrid Verbauwhede. Breaking Elliptic Curve Cryptosystems using Reconfigurable Hardware. *FPL 2010* pp. 133-138.
- [7] Tim Guneysu, Christof Paar, and Jan Pelzl. Special Purpose hardware for solving the Elliptic Curve Discrete Logarithm Problem. *ACM TRETTS 2008* vol. 1, issue 2. <http://doi.acm.org/10.1145/1371579.1371580>.
- [8] Bailey, D.V., Batina, L., Bernstein, D.J., Birkner, P., Bos, J.W., Chen, H.-C., Cheng, C.-M., Van Damme, G., de Meulenaer, G., Dominguez Perez, L.J., Fan, J., Guneysu, T., Gurkaynak, F., Kleinjung, T., Lange, T., Mentens, N., Niederhagen, R., Paar, C., Regazzoni, F., Schwabe, P., Uhsadel, L., Van Herrewege, A., Yang, B.-Y. Breaking ECC2K-130. 2009. <http://eprint.iacr.org/2009/541>
- [9] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation* 48(177):203-209, 1987.
- [10] I. Blake, G. Seroussi, and N. Smart. *Advances in Elliptic Curve Cryptography (London Mathematical Society Lecture Note Series)*. Cambridge University Press, New York, NY, USA, 2005.
- [11] V.S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology CRYPTO 85 Proceedings*, pages 417-426, 1986.
- [12] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918-924, 1978.
- [13] R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*. 36:627-630, 1981.
- [14] Tim Guneysu, Gerd Pfeiffer, Christof Paar, and Manfred Schimmler. 3 Years of Evolution: Cryptanalysis with COPACOBANA. *SHARCS'09* Lausanne, Switzerland. September 9-10, 2009.
- [15] P.C. van Oorschot and M.J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology '99*, vol.12 no.1 pp.1-28.
- [16] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*,. 48:243-264, 1987.
- [17] Piotr Majkowski, Mariusz Rawski, Tomasz Wojciechowski, Zbigniew Kotulski, Maciej Wojtyński. Heterogenic Distributed System for Cryptanalysis of Elliptic Curve Based Cryptosystems. *19th International Conference on Systems Engineering*, 2008. pp. 300-305.