Abstract—This paper reports a new speed record for FPGAs in cracking Elliptic Curve Cryptosystems. We conduct a detailed analysis of different $F_{2^m}$ multiplication approaches in this application. A novel architecture using optimized normal basis multipliers is proposed to solve the Certicom challenge ECC2K-130. We compare the FPGA performance against CPUs, GPUs, and the Sony PlayStation 3. Our implementations show low-cost FPGAs outperform even multicore desktop processors and graphics cards by a factor of 2.

Index Terms—Elliptic Curve Cryptography; Certicom Challenge; FPGA;

I. MOTIVATION: ATTACKING ECC2K-130

Elliptic-Curve Cryptosystems (ECC), independently invented by Miller [18] and Koblitz [15], are now commonplace in both the academic literature and practical deployments. ECC allow shorter key-lengths, ciphertexts, and signatures than other conventional cryptosystems, e.g., RSA, and thus admit valuable optimizations in computing and communication complexity.

The security of ECC relies on the difficulty of Elliptic Curve Discrete Logarithm Problem (ECDLP) [3]. Briefly speaking, ECDLP is to find an integer $n$ for two points $P$ and $Q$ on an elliptic curve $E$ such that $Q \equiv [n]P$. To use ECC in the real world, practitioners need to know: how big should the parameters (or, colloquially, the “key size”) be to avoid practical attacks? Choosing parameters too small allows computational attackers to solve the ECDLP instance, while choosing parameters too large wastes time, communication, and storage. To encourage investigation of these issues, researchers at Certicom Corp. published a list of ECDLP challenges in 1997 [7].

Smaller members of the list of Certicom ECDLP challenge problems have been solved. Escott et al. report on their successful attack on ECCp-97, an ECDLP in a group of roughly $2^{97}$ elements [8]. A larger instance, ECC2-109 was solved by Monico et al. [6]. Bos et al. analyze and solve the ECDLP in a group of roughly $2^{122}$ elements using PS3s [4].

This paper reports on our effort to solve one of the Certicom ECDLP challenge problems using FPGAs. We focus on a Koblitz curve challenge over $F_{2^{131}}$, called “ECC2K-130.” Compared to previous attacks, which are mostly implemented in software on general-purpose workstations, our work obtains a much higher performance-cost ratio by using FPGA platforms.

The rest of the paper is organized as follows. Section II summarizes related work. In Section III we give a short description of the function that we implement on FPGA. Section IV and V explore different algorithms and architectures. Section VI reports on the results, concluding in Section VII.

II. RELATED WORK

The strongest known attacks against the ECDLP are generic attacks based on Pollard’s rho method [20], [5]. Further improvements including parallelization and the use of group automorphisms were made by Wiener and Zuccherato [26], van Oorschot et al. [25], and Gallant et al. [9]. The parallelized Pollard rho method consists of parallel loops that search for distinguished points. Each loop starts from a random point on $E$ and ends when a distinguished point is hit. The core function is thus the iteration function. Our work implements this function along with its improvements.

FPGAs have been applied to the Pollard rho method in several previous works. Güneysu et al. analyze ECDLPs over fields of odd-prime characteristic [11], [10], targeting a machine with 128 low-cost FPGAs. They extrapolate that to break an ECDLP in a group of roughly $2^{131}$ elements using this machine as taking over a thousand years.

Similarly, Meurice de Dormale et al. apply FPGAs to the ECDLP [17]. Here, they use characteristic-two finite fields, but restrict their inquiry to polynomial basis. Although conventional wisdom has held that low-weight polynomial basis is a better choice, in our application we can take advantage of the free repeated squarings ($2^n$-th powers) offered in normal basis. In addition, recent progress on normal-basis multiplication by Shokrollahi et al. [24] and Bernstein and Lange [2] further improve the prospects for normal basis. Our work is the first
FPGA implementation of the parallelized Pollard rho method using normal basis multiplication.

This work is part of a global distributed effort to break the largest ECDLP ever solved [1]. While that paper summarizes the overall effort, here we focus on the FPGA implementation.

The Contribution of This Paper. As part of this effort, this paper explores FPGA implementation options for the core finite-field arithmetic operations as well as architectures. An in-depth comparison between multipliers (polynomial basis, Type-II normal basis, and Shokrollahi’s) is given. Most notably, this is the first FPGA implementation of Shokrollahi’s multiplication algorithm. Our work proves the superiority of an FPGA platform over other specialized architectures and its suitability for the tasks that are computationally demanding. Results in this paper are relevant both to the cryptographic community as well as those interested in fast cryptographic implementations in normal basis.

III. ITERATION FUNCTION

We briefly describe the iteration function which we implemented in the FPGAs in this section. The general attack strategy and the design rationale behind of the iteration function are found in [1].

Our condition for a point \( P_i(x, y) \) to be a distinguished point is that \( \text{HW}(x) \leq 34 \), where \( x \) is represented using type-II normal-basis and \( \text{HW}(x) \) returns the Hamming weight of \( x \). Our iteration function is defined as

\[
P_{i+1} = \sigma^j(P_i) \oplus P_i,
\]

where \( \sigma^j(P_i) = (x^{2^j}, y^{2^j}) \) and \( j = ((\text{HW}(x_{P_i})/2) \mod 8) + 3 \). To solve ECC2K-130, about \( 2^{60.9} \) iterations are expected in total [1].

An efficient implementation of the iteration function is thus the key step towards a fast attack. Given \( P_i(x, y) \), the iteration function computes \( P_{i+1}(x', y') \) using Eq.(1). Fig. 1 shows the data flow of the iteration function.

IV. OPTIMIZING FINITE-FIELD OPERATIONS

The iteration function consists of two multiplications, one inversion and several squarings in \( \mathbb{F}_{2^{m}} \). Thus, fast finite-field arithmetic is essential to optimize the attack.

A vast body of literature exists on finite field arithmetic, and we are free to choose from a variety of representations and algorithms. For example, we can use either polynomial basis or normal basis for element representation, and an iteration can be performed using either the Extended Euclidean Algorithm (EEA) or Fermat’s little theorem. This leads to an important question: which configuration (basis, multiplication algorithm, inversion algorithm) ensures the most efficient implementation of the aforementioned iteration function? We try to answer this question with complexity analysis and design-space exploration.

A. Multiplication

An element of \( \mathbb{F}_{2^{131}} \) can be represented in both polynomial basis \( \mathbb{P} \) and Type-II normal basis \( \mathbb{N} \), where

\[
\mathbb{P} = \{ 1, w, w^2, \ldots, w^{130} \},
\]

\[
\mathbb{N} = \{ \gamma + \gamma^{-1}, \gamma^2, \gamma^{-2}, \gamma^2 + \gamma^{-2}, \ldots, \gamma^{2^{130}}, \gamma^{-2^{130}} \}.
\]

Here \( w \) is a root of an irreducible polynomial of degree 131, while \( \gamma \) is a primitive \( 263^4 \) root of unity. Multiplication in polynomial basis has long been considered more efficient than normal basis. On the other hand, squaring in normal basis is simply a circular shift. Moreover, computing any power \( \alpha^{2^n} \) can be performed by circularly-shifting by \( n \) positions. We implemented both options for comparison.

Besides conventional multiplication algorithms in polynomial and normal basis, we also implemented a recently reported hybrid algorithm, due to Shokrollahi [24]. When multiplication is needed, two field elements are converted to polynomial basis, a polynomial-basis multiplication is carried out, then the results are converted back to normal basis and reduced. This paper includes the first FPGA implementation of this method.

1) Polynomial-Basis Multiplier: Algorithms for multiplication in polynomial basis consist of two steps, polynomial multiplication and modular reduction. They can be carried out separately or interleaved. Given two elements \( A(w) = \sum_{i=0}^{m-1} a_i w^i \) and \( B(w) = \sum_{i=0}^{m-1} b_i w^i \), a bit-serial modular multiplication algorithm is shown in Alg. 1.

It is well known that one way to reduce area complexity is to use a special-form polynomial \( P(w) \). For \( \mathbb{F}_{2^{131}} \) there exists a low-weight irreducible pentanomial \( P(w) = w^{131} + w^{13} + w^2 + w + 1 \). Thus, the complexity of step 3 in Alg. 1 is \( (m+4) \) XOR and \( (m+4) \) AND operations.

One can also compute \( C(w) = A(w)B(w) = \sum_{i=0}^{2^{m-2}} c_i w^i \) first, and then reduce it with \( P(w) \). In this case, the Karatsuba method [14] can be used to reduce the complexity of
Algorithm 1 Bit-serial modular multiplication in $\mathbb{F}_{2^m}$

Input: $A(w) = \sum_{i=0}^{m-1} a_i w^i$, $B(w) = \sum_{i=0}^{m-1} b_i w^i$ and $P(w)$.
Output: $A(w)B(w) \mod P(w)$.
1: $C(w) = \sum_{i=0}^{m} c_i w^i \leftarrow 0$;
2: for $i = m - 1$ to 0 do
3: \quad $C(w) \leftarrow w(C(w) + c_m P(w) + b_i A(w))$;
4: end for
Return: $C(w)/w$.

The complexity of a modular multiplication is $M$ requires around $P$.

We denote $A$ which leads to a hybrid normal-basis multiplication algorithm.

(Hand, it has higher complexity: $A$ can be implemented in a digit-serial manner. On the other hand, it has higher complexity: $2m$ AND and $2m$ XOR gates for an $m$-bit systolic multiplier.

3) Shokrollahi s multiplier: Shokrollahi discovered an efficient algorithm for basis conversion between permuted normal basis and polynomial basis. Later, Bernstein and Lange proposed further optimizations to this approach including a more straightforward conversion function. More details on this multiplication method can be found in their recent work [2]. The new polynomial basis ($n\mathbb{P}$) is defined in [2] as Type-II polynomial basis.

$$n\mathbb{P} = \{ (\gamma + \gamma^{-1}), (\gamma + \gamma^{-1})^2, \cdots, (\gamma + \gamma^{-1})^m \}$$

which leads to a hybrid normal-basis multiplication algorithm. We denote $A_{n\mathbb{P}}$ and $A_{p\mathbb{P}}$ the representation of $A$ using $n\mathbb{P}$ and $p\mathbb{P}$, respectively. A multiplication then proceeds as follows.

1) converting to polynomial basis: $A_{n\mathbb{P}} \leftarrow A_{p\mathbb{P}}$, $B_{n\mathbb{P}} \leftarrow B_{p\mathbb{P}}$;
2) polynomial multiplication: $C_{n\mathbb{P}} \leftarrow A_{n\mathbb{P}}B_{n\mathbb{P}}$;
3) converting back to normal basis ($2m$-bit conversion): $C_{p\mathbb{P}} \leftarrow C_{n\mathbb{P}}$;
4) reduction (folding).

Let $S_{p\mathbb{P}}$ be a transformation function that converts $A_{p\mathbb{P}}$ to $A_{n\mathbb{P}}$. The essential observation by Shokrollahi is that basis conversion can be recursively broken down to half-length transformations. Let $f_{p\mathbb{P}}$ and $f_{n\mathbb{P}}$ be corresponding representations of $f$ in $p\mathbb{P}$ and $n\mathbb{P}$, respectively.

$$f_{p\mathbb{P}} = \{ f_1, f_2, \cdots, f_8 \} \circ \{ (\gamma + \gamma^{-1})^2, (\gamma + \gamma^{-1})^4, \cdots, (\gamma + \gamma^{-1})^m \}$$

Converting $f_{p\mathbb{P}}$ to $f_{n\mathbb{P}}$ can be then performed with two 4-bit transformations:

$$\{ g_1, g_2, g_3, g_4 \} \mathbb{n} \leftarrow S_{p\mathbb{P}}(f_1, f_2, f_6, f_3 + f_5, f_4) \mathbb{pN}$$

$$\{ g_5, g_2, g_7, g_8 \} \mathbb{n} \leftarrow S_{p\mathbb{P}}(f_0, f_6, f_7, f_8) \mathbb{pN}$$

The $\mathbb{pN} \rightarrow \mathbb{nP}$ and $\mathbb{nP} \rightarrow \mathbb{pN}$ conversion in $\mathbb{F}_{2^m}$ takes $(m/2)\log_2(m/4)$ operations each. In total, one field multiplication takes $\mathbb{M}(m) + m \log_2 m + m \log_2(m/4)$ operations. A more detailed discussion on the complexity can be found in [2].

Based on the analysis above, we can draw the following conclusions:

- A bit-serial multiplier using polynomial basis has a lower area complexity than one using normal basis.
- When low-weight polynomials exist, Shokrollahi’s multiplication algorithm is likely to have a higher complexity than conventional polynomial basis multiplication since the base conversion step is more complex than the polynomial reduction.

Though it seems that polynomial basis should be used, normal basis offers several advantages in this specific application. First, the Pollard rho iteration function requires the Hamming weight of $x$-coordinate represented in normal basis. In fact, thanks to the Frobenius endomorphism, checking HW of $x$ in normal basis checks 131 points simultaneously. This brings a speedup of $\sqrt{131}$ to the attack [1]. Second, the iteration function includes two $\sigma^j(x) = x^{2^j}$ routines (known as $m$-squaring). In normal basis, $\sigma$ is essentially a circular shift of $j$ bits, and thus can be performed in one cycle. Gains in $m$-squaring compensate the loss in multiplications.

B. Inversion

Inversion is the most costly of the four basic field operations. The Extended Euclidean Algorithm (EEA) and Fermat’s
Algorithm 2: Simultaneous inversion (Batch size = 3)

Input: $\alpha_1, \alpha_2, \alpha_3$.
Output: $\alpha_1^{-1}, \alpha_2^{-1}$ and $\alpha_3^{-1}$.

1: $d_1 \leftarrow \alpha_1$
2: $d_2 \leftarrow d_1 \alpha_2$
3: $d_3 \leftarrow d_2 \alpha_3$
4: $u \leftarrow d_3^{-1}$
5: $t_3 \leftarrow ud_2$, $u \leftarrow u\alpha_3$
6: $t_2 \leftarrow ud_1$, $u \leftarrow u\alpha_2$
7: $t_1 \leftarrow u$

Return: $t_1, t_2, t_3$.

Little Theorem (FLT) are widely used to perform inversion. In polynomial basis, the binary variant of EEA is generally faster than FLT, but it requires dedicated hardware. On the other hand, FLT can be performed on standard multiplier, thus has almost no area overhead. In normal basis, the variant of FLT attributed to Itoh and Tsujii [13] is a better choice since squaring is free.

To further reduce the computation costs for inverses, we employ Montgomery’s trick that batches multiple inversions by trading inversions for multiplications [19]. Alg. 2 shows this method to invert three inputs. Indeed, we can trade one inversion for three extra multiplications. As a result, one iteration function uses $5M + (1/n)I$ where $n$ is the batch size.

Using this trick, a high performance inverter is no longer needed. Indeed, the delay caused by the inverter is $1/n$ cycles. It is negligible given a large $n$. Therefore, we choose FLT instead of EEA.

V. ARCHITECTURE EXPLORATION

The architecture of the engine has a fundamental impact on the overall throughput. Among all the design options the following two are of great importance.

1) Multiplier architecture
2) Memory architecture

As an architecture exploration, we implemented three different architectures using different types of multipliers.

A. Architecture I: Polynomial Multiplier

As a starting point, we take a programmable elliptic-curve coprocessor as the platform. A digit-serial polynomial multiplier (see [21] for details) is used. A dedicated squarer is included. In each loop, the $x$-coordinate is converted into its normal basis representation, and its Hamming weight is counted. This adds a base-conversion block and a Hamming-weight computation block.

On this platform, squaring or addition takes two clock cycles, while multiplication takes $[n/d] + 1$ cycles given a digit-size $d$. The design is synthesized using ISE 11.2 and the target FPGA is Xilinx Spartan-3 XC3S5000 (4FG676). Implementation results show that $d = 22$ gives the best trade-off in terms of area-delay product.

The design consumes 3,656 slices, including 1,468 slices for the multiplier, 75 slices for the squarer, 1,206 slices for the base conversion, 117 slices for Hamming weight calculation. One Pollard rho iteration takes 71 cycles; 35 cycles are used for multiplication. The design achieves a maximum clock frequency of 101 MHz, and one iteration takes 704 ns. The $m$-squaring is performed with $m$ successive squarings. Obviously, this architecture is not efficient. The $m$-squaring operations can be largely sped up when normal basis is used.

B. Architecture II: Type-II Normal Basis Multiplier

Archi-II uses a digit-serial normal basis multiplier. When $m$ is small, a full systolic architecture can be used, performing one multiplication per cycle. However, a systolic array for $F_{2^{131}}$ is too large (more than 20,000 slices on Spartan-3). Thus, a digit-serial architecture is used. Implementation results show that $d = 13$ gives the lowest area-delay product. The multiplier alone uses 2,093 slices.

The basis-conversion component in Archi-I is no longer needed in Archi-II, saving 1,468 slices. In total, the design uses 2,578 slices. On this platform, one Pollard rho iteration takes 81 cycles, including 55 cycles used for multiplication. Compared to Archi-I, the $m$-squaring operation is largely improved. However, the multiplier becomes much slower than that in Archi-I. The design achieves a maximum clock frequency of 125 MHz, and one iteration takes 648 ns.

C. Architecture III: Fully Pipelined Iteration Function

Archi-III unrolls the Pollard rho iteration such that a throughput of one iteration per cycle is achieved. Remember
that 5 multiplications are required for each iteration, as a result, five normal basis multipliers are used. The design is fully pipelined. Since additions and squarings are embedded in the pipeline, it increases the delay of one iteration but does not affect the throughput.

The implementation of Archi-III consumes 22,195 slices and 20 block RAMs (RAMB16s) on the Xilinx Spartan-3 XC3S5000 FPGA. One fully pipelined Shokrollahi’s multiplier uses 4,391 slices. The inverter itself uses 4,761 slices. In total, the design uses 26,731 slices. The post placing-and-routing results show that this design can achieve a maximum clock frequency of 111 MHz.

VI. RESULTS AND ANALYSIS

TABLE I: Cost for the iteration function using various architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Area #slice/BRAM</th>
<th>Freq. MHz</th>
<th>Delay Cycles per step</th>
<th>Throughput per FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archi-I: Polynomial basis</td>
<td>3,656 / 4</td>
<td>101</td>
<td>71</td>
<td>$12.8 \times 10^6$</td>
</tr>
<tr>
<td>Archi-II: Type-II ONB</td>
<td>2,578 / 4</td>
<td>125</td>
<td>81</td>
<td>$18.5 \times 10^6$</td>
</tr>
<tr>
<td>Archi-III: Shokrollahi’s</td>
<td>26,731 / 20</td>
<td>111</td>
<td>23</td>
<td>$111 \times 10^6$</td>
</tr>
</tbody>
</table>

* Note that Archi-III has 23 stages in the pipeline, thus has a delay of 23 cycles. But the average throughput is one iteration per cycle.

The ECC2K-130 attack using FPGAs is conducted using an improved version of the COPACOBANA cluster described in [10], [12], also known as RIVYERA [22]. It is populated with 128 Spartan-3 XC3S5000 FPGAs and an optional 32MB memory per FPGA combined in one 19” housing. All FPGAs are connected with two opposite directed, systolic ring networks that directly interface with the PC (which is integrated in the same housing) via two individual PCI Express communication controllers. Although this setup can obviously provide a significant amount of bandwidth due to its local communication paths, the ECC2K-130 attack design actually requires only moderate communication performance.

Table I summarizes the implementation results on a Spartan-3 XC3S5000 FPGA. Based on the available resources (33,280 slices and 104 BRAMs) of each XC3S5000 FPGA, we also estimated that at most 9 clones of Archi-I or 12 clones of Archi-II can be implemented on a single FPGA. For Archi-III, one clone uses 80% of the available resources of one FPGA.

The throughput per engine, $T_e$, is computed as $T_e = \frac{F_{\text{Freq}}}{c_{\text{cycles per step}}}$, and the throughput per FPGA $T_c$ is computed as $T_c = T_e \times l$. Here, $l$ is the number of engines on a single FPGA. Compared with Archi-I, Archi-II has smaller area and shorter delay. In other words, Type-II optimal normal basis has significant advantages for this application. On the other hand, Archi-III achieves a 8.6 times speedup over Archi-II. The improvement mainly comes from the efficient field arithmetic and the special architecture. The use of Shokrollahi’s algorithm significantly improved the throughput of a multiplier, while the expansion of the iteration function hides delays caused by addition and squaring in the pipeline.

In Table II we compare our results with similar implementations on different platforms.

To our knowledge, this is the first FPGA implementation using fast normal-basis multiplication to attack ECDLP. As a point of comparison, we look into the work of Meurice de
TABLE II: Performance comparison

<table>
<thead>
<tr>
<th>Source</th>
<th>Platform</th>
<th>Challenge</th>
<th>Freq. [MHz]</th>
<th>Throughput [×10^6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[17]</td>
<td>Xilinx FPGA S3E1200-4</td>
<td>ECC2K-130</td>
<td>100</td>
<td>10.0</td>
</tr>
<tr>
<td>[1]</td>
<td>Cell CPU 6 SPEs, 1 PPE</td>
<td>ECC2K-130</td>
<td>3,200</td>
<td>27.7</td>
</tr>
<tr>
<td>[1]</td>
<td>Core 2 Extreme Q6850, 4 cores</td>
<td>ECC2K-130</td>
<td>3,000</td>
<td>22.5</td>
</tr>
<tr>
<td>This work (Archi-III)</td>
<td>Xilinx FPGA XC3S5000</td>
<td>ECC2K-130</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Dormale, et al. [17]. They do not specifically target Koblitz curves and they are using different FPGAs, which makes a fair comparison difficult.

On the other hand, there is an interesting comparison between implementations of the same attack (and thus iteration function) on different platforms other than FPGAs. As part of the global distributed effort to attack ECC2K-130 [1], efforts have been made to speed-up the iteration function on CPUs, GPUs and PlayStation 3. These platforms are state-of-the-art processors supporting massive parallelism. Compared to these platforms, our FPGA implementation is at least 2 times faster in terms of throughput.

The whole complexity of this attack is around 2^{60.9} iterations. Populated with 128 FPGA each, a single COPACOBANA finishes 2^{58.7} = 128⋅(111⋅2^{20}⋅3600⋅24⋅365) iterations in one year. We estimate that given five COPACOBANA clusters, the ECC2K-130 challenge can be solved in one year.

VII. CONCLUSION

A new efficiency record for FPGAs in cracking public-key cryptosystems based on elliptic curves is reported. We conduct a detailed comparison of different architectures for normal-basis multipliers suited this application. The comparison includes the first FPGA implementation using normal basis. Our results show that even low-cost FPGAs outperform CPUs, the PlayStation 3 platform and even GPUs.

ACKNOWLEDGEMENTS

This work was supported in part by K.U. Leuven-BOF (OT/06/40), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by FWO project G.0300.07, and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

The authors would also like to thank Daniel J. Bernstein and Tanja Lange for insightful discussions and suggestions.

REFERENCES