IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 3, pp. 1–27. DOI:10.46586/tches.v2021.i3.1-27

Speed Reading in the Dark: Accelerating Functional Encryption for Quadratic Functions with Reprogrammable Hardware

Milad Bahadori[1], Kimmo Järvinen[1], Tilen Marc[2] and Miha Stopar[2]

[1] University of Helsinki, Department of Computer Science, Helsinki, Finland,
{milad.bahadori,kimmo.u.jarvinen}@helsinki.fi
[2] XLAB, Ljubljana, Slovenia, {tilen.marc,miha.stopar}@xlab.si

Abstract. Functional encryption is a new paradigm for encryption where decryption does not give the entire plaintext but only some function of it. Functional encryption has great potential in privacy-enhancing technologies but suffers from excessive computational overheads. We introduce the first hardware accelerator that supports functional encryption for quadratic functions. Our accelerator is implemented on a reprogrammable system-on-chip following the hardware/software codesign methogology. We benchmark our implementation for two privacy-preserving machine learning applications: (1) classification of handwritten digits from the MNIST database and (2) classification of clothes images from the Fashion MNIST database. In both cases, classification is performed with encrypted images. We show that our implementation offers speedups of over 200 times compared to a published software implementation and permits applications which are unfeasible with software-only solutions.

Keywords: Functional encryption · Hardware implementation · Privacy enhancing technologies · HW/SW codesign · FPGA · System-on-chip · Machine learning

1 Introduction

Functional Encryption (FE) [BSW11, O'N10] is a new paradigm for encryption where decryption does not return the original plaintext pt but only some function of it $f(\mathsf{pt})$. FE has great potential for improving privacy. E.g., users can encrypt their sensitive data so that a cloud server may compute statistics over the users' data without learning anything else about the data except the results of these computations. It is noteworthy that such solutions are not possible even with Fully Homomorphic Encryption (FHE) because FHE allows the cloud server to perform the required computations with the users' data, but the users' interaction would be needed to get the decrypted result.

Although theoretical FE constructions for arbitrary polynomial-sized circuits have been introduced (e.g., [Wat15, GGH+13]), they are not practical. (Semi-)practical FE constructions exist only for simple functions, namely, Inner Products (IP) (e.g., [ABDP15, ALS16, ACF+18]) and Quadratic Functions (QF) (e.g., [BCFG17, DGP18, Gay20, Wee20]). FE-IP allows computing (weighted) means over encrypted data sets and even simple Machine Learning (ML) models based, e.g., on linear regression. FE-QF opens up a significantly larger set of applications including more robust ML applications over encrypted data. An efficient FE-QF scheme was introduced by Dufour Sans et al. [DGP18] in the paper titled *"Reading in the Dark"* where they also demonstrated how it can be used for classifying handwritten digits from the well-known MNIST database[1] with encrypted images. Even

[1] Available in http://yann.lecun.com/exdb/mnist/

Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-07-09

FE-IP and, especially, FE-QF have practical limitations because computational overheads get excessive when data sets get large. Stopar et al. [MSH+19] recently published GoFE, an open source cryptographic library[2] written in Go language. The library supports multiple FE schemes including the FE-QF scheme from [DGP18]. They tested the FE-QF scheme for the MNIST database and stated that *"the decryption of one image [...] takes under 20 seconds"* which is still excessive for many practical applications.

Our objective is to show that the performance limitations of FE can be pushed significantly further by utilizing hardware acceleration. Specifically, we introduce an implementation for computing decryption of the FE-QF scheme from [DGP18] on a reprogrammable System-on-Chip (SoC) platform and demonstrate major speedups compared to a software implementation with GoFE. Our accelerator could be installed in a cloud server and it would enable significantly more complex privacy-preserving solutions than what are possible with software-only systems. To the best of our knowledge, the only published hardware acceleration result for FE was recently provided by Bahadori and Järvinen in [BJ20b]. They implemented a multi-input FE-IP scheme based on Paillier encryption from [ALS16, ACF+18] using Xilinx programmable SoC platforms.

Contributions In this paper, we provide the following contributions:

- We present the first hardware accelerator[3] for an FE-QF scheme. The accelerator is a hardware/software codesign implemented in a Xilinx reprogrammable SoC to maximally utilize the advantages of both software and hardware. The accelerator is optimized particularly for decryption. An FE-QF decryption consists of several cryptographic pairings followed by a discrete logarithm. The hardware side is implemented in programmable logic as a multi-core architecture where the cores are designed for efficient computation of cryptographic pairings and the discrete logarithm is implemented via an efficient interplay of software and hardware.

- We describe a parallel version of Shanks' baby-step giant-step discrete logarithm algorithm which is optimized for reasonably small positive and negative output values. The algorithm splits into precomputation and on-the-fly phases that compute and use a large precomputed table, respectively. We show that this algorithm has importance beyond our work by importing it into the GoFE software library where it provides significant speedups.

- We show that our accelerator provides large speedups compared to software-only implementations. In particular, we show that classification of handwritten digits from the MNIST database can be computed in 87 ms, representing over 200 times speedup compared to the original GoFE software library from [MSH+19] and 15.5 times speedup even against the optimized GoFE library that uses our algorithm for discrete logarithms. Our accelerator also opens up new applications that are out of the reach of published software-only solutions such as accurate classification of clothes images from the Fashion MNIST database[4]. The model that we use in this classification is arguably too complicated to be practical with software as even the optimized GoFE library requires several seconds for a single classification—a task that our accelerator solves in only 0.38 seconds.

Paper organization The remainder of the paper is structured as follows. Sect. 2 introduces the preliminaries of FE, cryptographic pairings, and the specific FE-QF scheme that is implemented in this paper. Sect. 3 describes the architecture of the accelerator and Sect. 4

[2] Available in https://github.com/fentec-project/gofe
[3] Available in https://github.com/fentec-project/fe-qf-hardware-accelerated
[4] Available in https://www.kaggle.com/zalando-research/fashionmnist

shows how the FE-QF decryption is mapped into this architecture. Sect. 5 provides implementation results and general performance comparisons against software libraries. Sect. 6 introduces the use cases, provides the results for them, and compares the results with software implementations. Finally, Sect. 7 ends the paper by drawing conclusions.

2 Preliminaries

This section provides the required preliminaries for the rest of the paper. We present the background for FE and cryptographic pairings in Sect. 2.1 and Sect. 2.2, respectively. In Sect. 2.3, we describe the FE-QF scheme from [DGP18] that we implement in this paper.

2.1 Functional Encryption

Traditional encryption is all-or-nothing because decryption returns the entire plaintext pt from the ciphertext ct and nothing can be learned about the contents without the correct decryption key dk. FE [BSW11, O'N10] provides more fine-grained control. FE allows to derive decryption keys dk_f for different functions f so that dk_f enables to compute the value of $f(\mathsf{pt})$ from ct without giving any additional information about pt. Many potential applications of FE include spam filtering from encrypted email, facial recognition from encrypted photos or video, privacy-preserving medical data processing, etc.

While such applications are theoretically possible with the existing FE constructions for arbitrary polynomial-sized circuits (e.g., [Wat15, GGH+13]), they are not practical as these FE constructions are extremely inefficient. Abdalla et al. [ABDP15] chose to take a different, more practical approach where FE schemes are designed for simple but still practically meaningful functions. This trend of research has resulted in many schemes for IP (e.g., [ABDP15, ALS16, ACF+18]), i.e. functions of the form: $\langle \mathbf{f}, \mathbf{x} \rangle = \sum_{i=0}^{n-1} f_i x_i$. Baltico et al. [BCFG17] proposed the first efficient FE scheme for other than linear functions. Their FE-QF scheme supports functions of the form $\mathbf{x}^\top \mathbf{F}\, \mathbf{y} = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} f_{i,j} x_i y_j$. Dufour Sans et al. presented a slightly more efficient FE-QF in [DGP18]. The latter plays the central role in our work and is described with more details in Sect. 2.3.

The support for more complex functions increases the applicability of FE in practical applications. FE-IP computes (weighted) sums of inputs and, consequently, supports applications like voting (e.g., sums), basic statistics (e.g., weighted average) or simple machine learning (e.g., linear regression). Thanks to the ability to compute sums of products of inputs, FE-QF additionally supports applications like variance and correlation in statistics or simple neural networks in machine learning enabling better prediction accuracy. The use cases considered in Sect. 6 are examples of the capabilities of FE-QF, but the applications are certainly not limited to them.

2.2 Cryptographic Pairings

A cryptographic pairing is a bilinear map $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ from two additive groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to a multiplicative group $\mathbb{G}_3$. The first use of bilinear pairings in cryptology was for cryptanalysis [MOV93]. Later they have been utilized for multiple cryptographic constructions including tripartite key exchange [Jou04], identity-based encryption [BF01], short signatures [BLS04], attribute-based encryption [GPSW06, BSW07], searchable encryption [ABC+08], and—most importantly for this work—FE [BCFG17, DGP18, AGRW17]. These applications need efficiently computable pairings and research for improving efficiency has resulted in various types of pairing algorithms and pairing-friendly elliptic curves such as the Barreto-Naehrig (BN) curves [BN06]. In this paper, we use optimal ate pairings [Ver10] on BN curves as proposed by Beuchat et al. in [BGM+10]. In that case, $\mathbb{G}_1$ and $\mathbb{G}_2$ are points on elliptic curves $E(\mathbb{F}_p)$ and $E(\mathbb{F}_{p^2})$ and $\mathbb{G}_3$ is the multiplicative

group of $\mathbb{F}_{p^{12}}$ such that p is a 254-bit prime. Due to Kim and Barbulescu's extended number field sieve algorithm [KB16], it is considered to offer over 100-bit security.

2.3 Functional Encryption for Quadratic Functions from Pairings

Next, we describe the FE-QF scheme introduced by Dufour Sans, Gay, and Pointcheval in [DGP18]. The scheme is based on cryptographic pairings and is secure under the generic group model. It supports QF $f(\mathbf{x}, \mathbf{y})$ parametrized with $n, B_x, B_y, B_f \in \mathbb{N}^*$ so that $f(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_{i,j} x_i y_j$ where $x_i \in [-B_x, B_x]$, $y_j \in [-B_y, B_y]$, and $f_{i,j} \in [-B_f, B_f]$. The scheme defines four routines:

Set-up generates a public key pk and a master secret key msk. It takes a bilinear pairing scheme $\mathcal{PG} = (\mathbb{G}_1, \mathbb{G}_2, p, g_1, g_2, e)$ for a security parameter 1^λ and the maximum values for the above function parameters as inputs. It returns a public key $\mathsf{pk} = (g_1^{\mathbf{s}}, g_2^{\mathbf{t}})$ and a master secret key $\mathsf{msk} = (\mathbf{s}, \mathbf{t})$, where $\mathbf{s}, \mathbf{t}$ are n random elements of $\mathbb{Z}_p$.

Encryption encrypts the data vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ where $x_i \in [-B_x, B_x]$ and $y_j \in [-B_y, B_y]$. It selects a random $\gamma \in \mathbb{Z}_p$ and a random 2×2 invertible matrix $\mathbf{W}$ and computes $\mathbf{a}_i = (\mathbf{W}^{-1})^\top (x_i, \gamma s_i)^\top$ and $\mathbf{b}_i = \mathbf{W} (y_i, -t_i)^\top$. Then, it returns the ciphertext $\mathsf{ct} = \left(g_1^\gamma, \{g_1^{\mathbf{a}_i}, g_2^{\mathbf{b}_i}\}_{i \in [n]}\right)$.

Key Generation generates a decryption key dk_f for a specific f, for which $f_{i,j} \in [-B_f, B_f]$, by using msk. Specifically, it computes and outputs $\mathsf{dk}_f = \left(g_2^{f(\mathbf{s},\mathbf{t})}, f\right)$.

Decryption evaluates the function f on the ciphertext ct using the decryption key dk_f and the public key pk and returns an integer value $r = f(\mathbf{x}, \mathbf{y})$. It computes

$$\rho = e(g_1^\gamma, g_2^{f(\mathbf{s},\mathbf{t})}) \cdot \prod_{i,j \in [n]} e(g_1^{\mathbf{a}_i}, g_2^{\mathbf{b}_i})^{f_{i,j}} \tag{1}$$

where e is the bilinear pairing. The final result r is obtained by solving the discrete logarithm $r = \log_{g_3}(\rho)$ where $g_3 = e(g_1, g_2)$. Because we focus particularly on the decryption routine in this paper, we describe it with more details in Alg. 1.

The details about the correctness and security of the scheme are presented in [DGP18].

As can be seen above, the encryption and key generation routines consist mainly of scalar multiplications in the elliptic curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with base points g_1 and g_2, respectively. Decryption divides in two parts: (a) $2n^2 + 1$ cryptographic pairing computations and (b) one discrete logarithm in the finite field $\mathbb{G}_3 = \mathbb{F}_{p^{12}}$. In addition

Algorithm 1: Decryption of the FE-QF scheme [DGP18]

Input: $\mathsf{ct} = (g_1^\gamma, \{g_1^{\mathbf{a}_i}, g_2^{\mathbf{b}_i}\}_{i \in [n]})$, $\mathsf{dk}_f = (g_2^{f(\mathbf{s},\mathbf{t})}, f)$, $g_3 = e(g_1, g_2)$
Output: $r \in \mathbb{Z}$

1 $\rho \leftarrow e(g_1^\gamma, g_2^{f(\mathbf{s},\mathbf{t})})$ // Pairing
2 **for** $i = 0, \ldots, n-1$ **do**
3 **for** $j = 0, \ldots, n-1$ **do**
4 $e_0 \leftarrow e(g_1^{a_{i,0}}, g_2^{b_{j,0}})$ // Pairing
5 $e_1 \leftarrow e(g_1^{a_{i,1}}, g_2^{b_{j,1}})$ // Pairing
6 $\rho \leftarrow \rho \cdot (e_0 \cdot e_1)^{f_{i,j}}$ // Inv. (if $f_{i,j} < 0$), exp., and two mults.
7 $r \leftarrow \log_{g_3}(\rho)$ // Discrete logarithm

Algorithm 2: Optimal ate pairing over BN curves [BGM+10]

Input: $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$.
Output: $e(Q,P) = f$, where $f \in \mathbb{F}_{p^{12}}$.
Constant: $s = 6t+2 = \sum_{i=0}^{L-1} s_i\, 2^i$, where $t = 2^{62} - 2^{54} + 2^{44}$ and $s_i \in \{0, \pm 1\}$.

1 $T \leftarrow Q$; $f \leftarrow 1$
2 **for** $i = L-2$ **to** 0 **do**
3 $\quad f \leftarrow f^2 \cdot l_{T,T}(P)$; $T \leftarrow 2T$
4 $\quad$ **if** $s_i \neq 0$ **then** $f \leftarrow f \cdot l_{T,s_iQ}(P)$; $T \leftarrow T + s_i\, Q$
5 $Q_1 \leftarrow \pi_p(Q)$; $Q_2 \leftarrow -\pi_{p^2}(Q)$
6 $f \leftarrow f \cdot l_{T,Q_1}(P)$; $T \leftarrow T + Q_1$
7 $f \leftarrow f \cdot l_{T,Q_2}(P)$; $T \leftarrow T + Q_2$
8 **return** $f^{(p^{12}-1)/r}$

to these, Alg. 1 requires n^2 exponentiations (typically with small exponents) and $2n^2$ multiplications in $\mathbb{F}_{p^{12}}$ as well as one inversion in $\mathbb{F}_{p^{12}}$ for each $f_{i,j} < 0$. However, the cost of these operations is insignificant compared to the pairings and the discrete logarithm.

The function f is applied in Alg. 1 by computing exponentiations with $f_{i,j}$ in line 6. If $f_{i,j} = 0$, then the result of this exponentiation is $1 \in \mathbb{F}_{p^{12}}$ and the multiplications and the optional inversion in line 6 are not needed. Even more importantly, the pairings in lines 4 and 5 can be skipped too. For this reason, decryption with sparse functions are significantly cheaper operations than general decryptions. This has been utilized extensively in [DGP18, MSH+19] where image classifications utilize projections that result in f where only the diagonal values ($i = j$) are nonzero. We utilize similar projections in the use cases of Sect. 6 and compute functions of the form $f(\mathbf{x},\mathbf{x}) = \sum_{i=0}^{n-1} \hat{f}_i x_i^2$ where $\hat{f}_i$ is the i^{th} value of the diagonal reducing the number of required operations per decryption to only $2n+1$ pairings and one discrete logarithm in $\mathbb{F}_{p^{12}}$ (and $2n$ multiplications, n exponentiations, and one inversion for each negative $\hat{f}_i$). However, our implementation is not limited to such structure of f and we provide performance results for both general and diagonal f in Sect. 5.

We instantiate the FE scheme of [DGP18] with optimal ate pairings over BN curves as described by Beuchat et al. [BGM+10] and Shanks' baby-step giant-step discrete logarithm algorithm, which are discussed in Sects. 2.3.1 and 2.3.2, respectively.

2.3.1 Optimal Ate Pairing over a Barreto-Naehrig Curve

The optimal ate pairing over BN curves introduced by Beuchat et al. in [BGM+10] is given in Alg. 2. The main operations in Alg. 2 are the Miller loop in lines 2–4 and the final exponentiation in line 8. The Miller loop consists of elliptic curve point additions and doublings in $E(\mathbb{F}_{p^2})$ and line evaluations in $\mathbb{F}_{p^{12}}$. The final exponentiation is the computation of $f^{(p^{12}-1)/r}$ in $\mathbb{F}_{p^{12}}$ that can be decomposed into $f^{(p^6-1)(p^2+1)(p^4-p^2+1)/r}$ as shown by Scott et al. in [SBC+09]. The decomposition allows significantly faster computation because the two first terms can utilize Frobenius operators and conjugations. The last term $f^{(p^4-p^2+1)/r}$ is the hard part, but it can be computed more efficiently with a vectorial addition chain as shown in [SBC+09]. We refer the readers to [BDM+10] for detailed algorithms (31 in total) for computing each suboperation of Alg. 2.

2.3.2 Discrete Logarithms with Shanks' Baby-Step Giant-Step Algorithm

Baby-step giant-step algorithm given in Alg. 3 is a classical meet-in-the-middle algorithm invented by Daniel Shanks in the early-1970s for computing discrete logarithms in

Algorithm 3: Shanks' baby-step giant-step algorithm for discrete logarithms [Sha71]

Input: $\alpha, \beta \in \mathbb{G}$ such that $\beta = \alpha^x$ with $x \in \mathbb{Z}$ and $\mathbb{G}$ is a cyclic group of order ν
Output: $x = \log_\alpha(\beta) \in \{1, \ldots, \nu - 1\}$

$\mu \leftarrow \lceil \sqrt{\nu} \rceil$; $\gamma \leftarrow 1$
for $1 \leq j < \mu$ **do**
  $\gamma \leftarrow \gamma \cdot \alpha$
  Add (j, γ) to the table T
$\alpha' \leftarrow \alpha^{-\mu}$; $\gamma' \leftarrow \beta$
for $0 \leq i < \mu$ **do**
  if $\exists (j, \alpha^j) \in T$ *such that* $\alpha^j = \gamma'$ **then return** $x = i \cdot \mu + j$
  $\gamma' \leftarrow \gamma' \cdot \alpha'$

groups [Sha71]. When given α and β such that $\alpha, \beta \in \mathbb{G}$ with an order ν and $\beta = \alpha^x$, it returns $x \in [0, \nu - 1]$. The algorithm works in two phases. Firstly, a table T is constructed comprising tuples (j, α^j) for all powers $j = 0, 1, \ldots, \mu - 1$ where $\mu = \lceil \sqrt{\nu} \rceil$. This is the so-called baby-step phase. Secondly, it makes trials by computing $\beta \cdot (\alpha^{-\mu})^i$ for $i = 0, 1, \ldots$ until it finds a match in T. This is the giant-step phase. When a match is found, then $x = i \cdot \mu + j$ because $\alpha^j = \beta(\alpha^{-\mu})^i$ implies $\beta = \alpha^{i \cdot \mu + j}$.

Instead of a general discrete logarithm in $\mathbb{G}_3 = \mathbb{F}_{p^{12}}$, the FE-QF scheme requires discrete logarithms where the outputs are bounded to $[-B_t, B_t]$ where $B_t = n^2 \cdot B_x \cdot B_y \cdot B_f$ such that $B_t \ll \nu$. Consequently, the discrete logarithm problem can be solved in reasonable time. Because g_3 in Alg. 1 is a fixed domain parameter ($g_3 = e(g_1, g_2) \in \mathbb{F}_{p^{12}}$), the table T can be precomputed. Sect. 4.5 describes how Alg. 3 is computed in our implementation.

3 Architecture

This section describes the architecture of our accelerator. It is a HW/SW codesign, where the computationally heavy cryptographic pairings and $\mathbb{F}_{p^{12}}$ arithmetic are supported by the HW domain (programmable logic). The SW domain controls the HW domain and computates auxiliary operations. In many settings, the HW/SW codesign paradigm allows combining high performance with low resource usage and flexibility. This is certainly true for implementing the FE-QF scheme which must support a myriad of separate algorithms (i.e., suboperations of cryptographic pairings, arithmetic in $\mathbb{F}_{p^{12}}$, and discrete logarithms) so that computations still mostly rely on the same low-level arithmetic in $\mathbb{F}_p$.

The FE-QF decryption includes a lot of inherent parallelism. Consequently, we design the HW domain as a multi-core architecture including multiple parallel and programmable Cryptography Processor (CP) cores. We base our CP cores on the pairing cryptography processor architecture recently published in [BJ20a] because it has a good balance between performance and area requirements and is already designed for a HW/SW codesign in a reprogrammable SoC. We instantiate a multi-core architecture where each CP core can be programmed with different microprograms and, hence, the architecture supports two types of processing: (a) symmetric processing, where the CP cores process the same computations (but with different data), and (b) asymmetric processing, where the CP cores process different computations. Implementation of the FE-QF scheme requires significant amounts of data and microcode transfers between the HW and SW domains that easily become the bottleneck for performance. We mitigate this limitation by introducing different levels of data and program memories in the HW domain to reduce HW/SW communication. These memories include both global (for all CP cores) and local (for a single CP core) data and program memories. In the SW domain, we utilize both on-chip and off-chip memories.

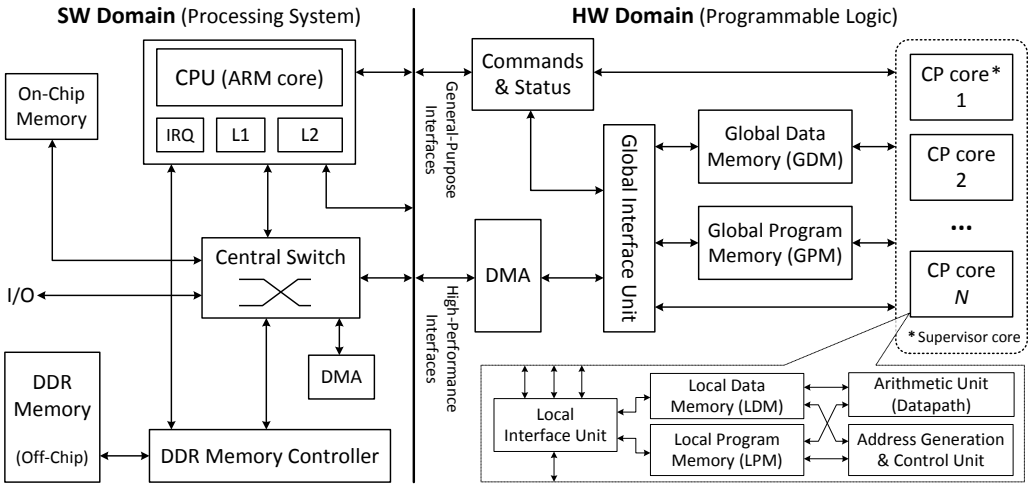

Figure 1: High level architecture of the programmable HW/SW codesign accelerator.

Also other HW/SW codesigns for pairings than [BJ20a] could be potentially used, such as [SDK17]. However, they may require adaptation of slightly different approaches in the codesign; e.g., [SDK17] presents a slower and smaller core compared to [BJ20a] and would require a much larger number of parallel cores for similar performance. Certain stand-alone coprocessors for pairings such as [SGZ+15, YFCV13] allow faster pairings than [BJ20a] but they are typically much larger and would not permit computation of other operations needed in FE-QF. Consequently, we focus solely on the use of [BJ20a] in this paper.

3.1 High-Level Architecture of HW/SW Codesign

Fig. 1 illustrates the high-level architecture of the HW/SW codesign which is divided into two main parts: (1) SW domain and (2) HW domain. The architecture is generic in the sense that it can be instantiated in various programmable SoCs with minor modifications (see App. A for additional discussion). However, in this paper, we consider only instantiations in Xilinx all-programmable SoCs because we use a Xilinx Zynq Ultrascale+ multiprocessor SoC (MPSoC) platform for prototyping. In an MPSoC, the SW domain consists of multi-core ARM processors and the HW domain is a powerful Field Programmable Gate Array (FPGA). To provide programmability and to decrease resource utilization, the HW domain utilizes a microprogramming approach instead of hardwired Finite State Machines (FSMs).

3.1.1 SW Domain

The SW domain (on the left in Fig. 1) is responsible for controlling all operations in the HW domain and external peripherals (i.e., DDR memory and I/O peripherals) as well as performing computations which are not supported by the multi-core architecture of the HW domain. The SW domain controls all actions in the HW domain including sending and receiving data and microprogram packs between the SW and HW domains, issuing/receiving commands/statuses to/from the HW domain, configuring and programming all CP cores and other modules in the HW domain, and making control decisions for symmetric or asymmetric parallel execution of the CP cores based on the received statuses and different conditions. The communication between the SW and HW domains is performed via two types of interfaces: High Performance (HP) and General Purpose (GP) interfaces. The HP interfaces are employed for high performance data and microprograms communications whereas the GP interfaces are used for transfering commands and statuses.

3.1.2 HW Domain

The HW domain (on the right in Fig. 1) contains the parallel CP cores for performing the actual computations and many supporting modules for data and microprograms communication and storage as well as for commands and status communication. All modules in the HW domain are connected to the Advanced Extensible Interface (AXI) structure. The main challenge is communicating data and microprogram packs between the SW and HW domains so that it does not become a bottleneck for performance. To alleviate this limitation, various global and local data and program memories are used in the HW domain. As mentioned above, this communication uses the HP interfaces that connect to the global interface unit via an AXI Direct Memory Access (DMA) module. The global interface unit in the HW domain is controlled by the command and status signals and is connected to the Global Data Memory (GDM), Global Program Memory (GPM), as well as directly to the parallel CP cores. It is responsible for managing all data and microprogram transactions between the SW domain memories, global memories, and local memories of the CP cores. Each CP core has its own Local Data Memory (LDM) and Local Program Memory (LPM). Memory transactions for data and microprograms can be done between (1) the SW domain and the global memories of the HW domain, (2) the SW domain and the local memories of the HW domain, and (3) the global and local memories of the HW domain. Fig. 1 also shows the simplified architectural diagram of the CP core which we discuss with more details in Sect. 3.2.

The multi-core architecture contains N parallel CP cores and supports two types of execution flows: (a) symmetric and (b) asymmetric parallel execution. In the symmetric parallel execution, each CP core uses its own LDM and the shared GPM. One CP core (i.e., CP core 1 in Fig. 1) works as the supervisor core and interacts with the GPM by fetching the required microprograms. All CP cores (i.e., CP cores $1, 2, 3, \ldots, N$) then execute the fetched instructions in a symmetric manner so that the same microprograms are executed in parallel with different data. In the asymmetric parallel execution, each CP core works independently using its own LPM and LDM. As the result, all CP cores can execute different microprograms in parallel. The GDM is a global shared memory for storing input, output, and intermediate data. It allows fast data transfers between the CP cores without the need to move the data between the HW and SW domains.

3.2 Cryptography Core

The architecture of the CP core is based on the pairing cryptography processor described in [BJ20a]. The main idea behind its design was to achieve a good trade-off between speed and area requirements by optimizing the core for the resources of modern FPGAs and by using the HW/SW codesign paradigm. This is in contrast with many other cores available in the literature which have been designed to minimize the latency of a single pairing computation with an expense of a larger resource usage. Having a core that is optimized for the speed-area tradeoff and the HW/SW codesign paradigm provides an excellent starting point for designing an efficient multi-core architecture for FE-QF. The CP core from [BJ20a] was considered to be implemented primarily as a stand-alone core, but our architecture requires its use in a multi-core architecture and, consequently, we introduce certain changes into the architecture of the CP core and its interfaces.

The CP core is based on a microprogramming architecture in order to combine runtime programmability with a small area footprint. The optimal ate pairing of Alg. 2 and the discrete logarithms with Alg. 3 both rely on $\mathbb{F}_{p^{12}}$ arithmetic which is ultimately based on $\mathbb{F}_p$ arithmetic. Hence, the efficiency of the entire FE-QF scheme strongly depends on the efficiency of $\mathbb{F}_p$ arithmetic and the scheduling of these arithmetic operations.

Although the CP core is based on the core in [BJ20a], we modified and extended the original architecture in multiple ways in particular to facilitate its efficient use in the

multi-core architecture. We changed its LDM and LPM units as well as its local interface unit to support local and global modes required for symmetric and asymmetric processing. The architecture details of the CP core are depicted in Fig. 2, which contains local interface, arithmetic (datapath), the LDM, the LPM, and address generation and control units. The local interface unit communicates commands, statuses, data, and microprograms with the global interface unit and other HW modules. This unit loads data into the LDM and microprograms into the LPM. The latter can be done in two ways: offline before the actual FE-QF computations or on-the-fly during the computations. The LPM contains a simple dual-port RAM and a controller for different address branch scenarios. The LPM stores microprograms for algorithm(s) that are run in the CP core. Each instruction in a microprogram consists of several fields providing commands to the corresponding units for a working cycle of the CP core. The LPM is partitioned into several segments, where each segment can be loaded separately via the local interface unit during the runtime. In addition, full microprogram loading of the LPM can be done directly from the SW domain or from the GPM of the HW domain.

The control unit generates addresses for the LDM and makes decisions for loop iterations and conditional statements. The inputs and outputs of the arithmetic unit are connected to the LDM. The LDM is a duplicated true dual-port RAM with two independent read and write ports and supports "4-read", "2-write", or "2-read and 1-write" operations from/to the LDM. This facilitates efficient scheduling and parallelization of $\mathbb{F}_p$ arithmetic. The LDM is also interfaced with the local interface unit for communicating data with the global interface unit (i.e., data communications with the GDM and the SW domain).

The arithmetic unit (datapath) is described in Fig. 2. In this work, the datapath width is 256-bit and it supports up to 256-bit arithmetic computations in $\mathbb{F}_p$. It consists of three parts: source registers, arithmetic blocks, and output selectors. The arithmetic blocks comprise three Montgomery Modular Multiplier Blocks (MMMBs) and two Modular Adder/Subtractor Blocks (MASBs) and they can operate in parallel and independently of each other. The inputs of all arithmetic blocks can be loaded from the LDM but the inputs of the MASBs can be additionally loaded from the outputs of the arithmetic blocks. This arrangement together with the multi-read/write feature of the LDM allows efficient computation of $\mathbb{F}_{p^{12}}$ arithmetic. The modulus and the precomputed constant for the Montgomery arithmetic are registered into the arithmetic unit. Because the arithmetic unit is similar to the one in [BJ20a], we provide a brief summary of its structure in App. B and refer to [BJ20a] for a detailed description.

4 Implementation of the Algorithms

This section describes how the algorithms of the FE-QF scheme are mapped into the architecture introduced in Sect. 3. The entire mapping is done by writing the required software for the SW domain and the microprograms for the HW domain. In general, the mapping of the pairing algorithm is done following the guidelines of [BJ20a]. The discrete logarithm computations and parallel processing of the pairings and $\mathbb{F}_{p^{12}}$ arithmetic are novel contributions of this paper. Hence, the focus of this section is on these topics.

4.1 Working Principles of the HW/SW Codesign Accelerator

Fig. 3 explains the working principles and memory taxonomy of the SW and HW domains. It describes the details of different memories in the SW and HW domains as well as how they are utilized in the HW/SW codesign. Fig. 3 also shows the interaction principles including the formats of the instruction set of the CP core and the command and status signals. The HW/SW codesign must be initialized at the boot-up time. The initialization step must be done only once for each parameter set. It configures the SW and HW

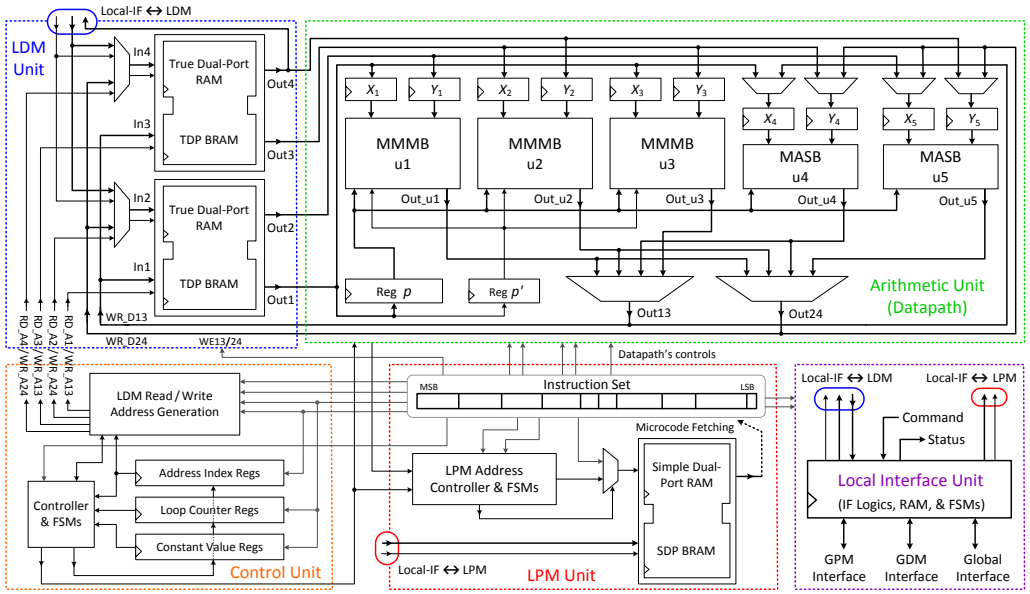

Figure 2: Architecture details of the CP core.

domains with the specific parameters and algorithms of the FE scheme and performs the precomputation for the discrete logarithm.

The algorithms are implemented via optimized microprograms which divide into two types: segment and full microprogram packs. The former allows updating only parts of the microprogram that is currently in the HW domain. To implement a specific subalgorithm of Alg. 1, an indepth analysis was performed and the algorithms were translated into microprograms (i.e., several segments and/or full sub-routine packs). The microprograms were generated by hand through a customized platform and scripts. The microprograms are sequences of instructions for different units of the CP core. The instructions are 72 bits long and divide into 14 fields (e.g., arithmetic, control, next program memory address, LDM address values, LDM, and LPM fields). All microprograms required for the FE-QF decryption are stored in the (off/on-chip) SW domain memory (i.e., DRR memory). Whenever a (set of) particular computation(s) needs to be executed in a specific CP core or in all CP cores, the corresponding microcode(s) are loaded into LPM or GPM, as explained before. Finite field arithmetic has the largest impact on the overall performance and, therefore, special care was taken to optimize microprograms for them to maximimally utilize the datapath of the CP cores (see Sect. 4.2).

In the HW domain, the (full and segment) microprogram packs are stored in the LPM and/or GPM memories depending on how they need to be executed. In this work, the sizes of segment and full packs are 288 B and 18 KB, respectively. In symmetric parallel execution, all microprogram packs are first stored in the GPM and the supervisor core (i.e., CP core 1) interacts with the GPM to get the suitable packs for all CP cores. In asymmetric execution, the SW domain stores different microprogram packs into the LPMs of different CP cores. The latencies reported in the following sections include all latencies related to transferring microprograms, commands, and statuses between the SW and HW domains and also between different modules inside the HW domain. The SW domain contains a large off-chip memory (i.e., a 4 GB DDR memory) and most of this memory is occupied by a large precomputed table for discrete logarithm computations (see Sect. 4.5).

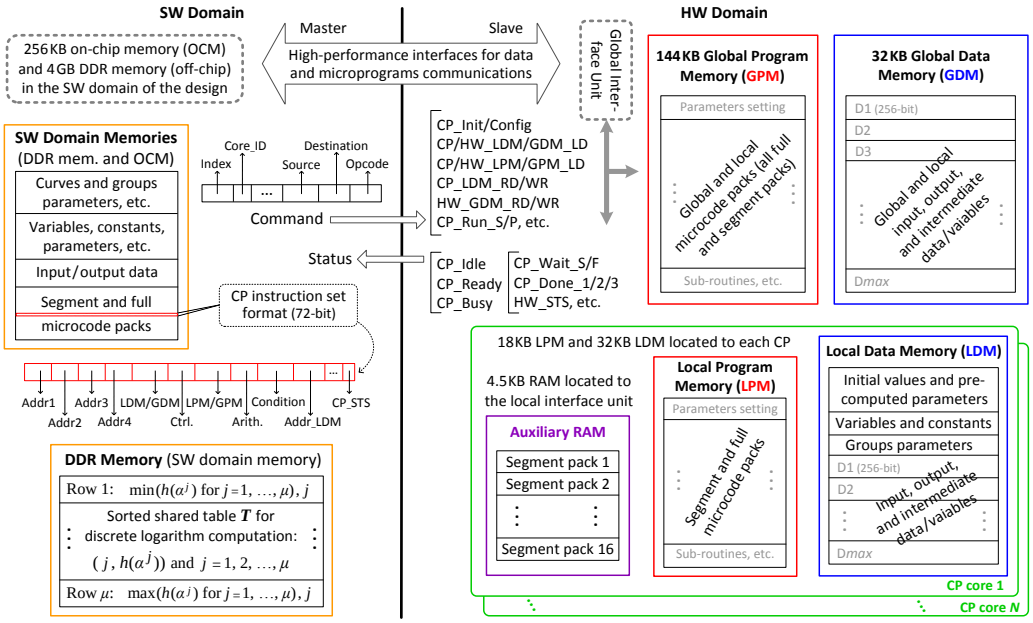

Figure 3: Memory taxonomy and interaction principles of the SW and HW domains.

4.2 Tower Extension Field Arithmetic

The first step in implementing the FE-QF scheme is to efficiently implement the finite field arithmetic in $\mathbb{F}_{p^{12}}$. In this work, we adopt the tower extension field definitions for $\mathbb{F}_{p^{12}}$ from [BDM+10] and, consequently, arithmetic operations are computed with series of operations in $\mathbb{F}_p$. In particular, Karatsuba-like multiplications in the quadratic extension fields $\mathbb{F}_{p^2}$ and $\mathbb{F}_{p^{12}}$ can be computed with three multiplications (and additions/subtractions) in $\mathbb{F}_p$ and $\mathbb{F}_{p^6}$, respectively. Multiplications in $\mathbb{F}_{p^6}$ require six multiplications in $\mathbb{F}_{p^2}$ (see [BDM+10] for detailed algorithms). As our CP core is based on the pairing cryptography processor from [BJ20a], we use the same approach for scheduling the tower extension field arithmetic (from $\mathbb{F}_p$ to $\mathbb{F}_{p^{12}}$). The main observation was that the parallel processing capabilities of the datapath can be utilized so that costs of all additions/subtractions are effectively hidden as they are executed in parallel with multiplications. Fig. 9 in App. B presents the timing diagrams and [BJ20a] provides additional details.

4.3 Optimal Ate Pairings over BN Curves

Implementation of optimal ate pairings (Alg. 2) consists of three main levels. The first level is the finite field arithmetic (from $\mathbb{F}_p$ to $\mathbb{F}_{p^{12}}$) discussed in Sect. 4.2. The second level are the elliptic curve doubling/addition steps and line evaluation functions (i.e., lines 3, 4, 6, and 7) as well as Frobenius operators (i.e., line 5). Finally, the third level controls the high-level operations of Alg. 2, which are the Miller loop (lines 2–4), Frobenius operators and final addition steps (lines 5–7), and final exponentiation (line 8). All levels are implemented by using the algorithms from [BDM+10]. The entire implementation contains 8 full and 24 segment microprogram packs. The Miller loop, Frobenius operators and final addition steps, and final exponentiation consist of 1/12, 1/2, and 6/10 full/segment packs, respectively. Alg. 2 contains 8 parts (and as many full packs). The Miller loop and lines 5–7 are the two first parts. The final exponentiation divides into 6 consecutive parts, which are computed following the procedure in [SBC+09] and the specific algorithms from [BDM+10]. Alg. 2 needs, in total, 14681 multiplications/squarings in $\mathbb{F}_p$.

4.4 Parallel Pairings and $\mathbb{F}_{p^{12}}$ Arithmetic Calculations

As mentioned in Sect. 2.3, the FE-QF decryption algorithm given in Alg. 1 is divided into two parts: (1) parallel pairings and $\mathbb{F}_{p^{12}}$ computations and (2) discrete logarithm in $\mathbb{G}_3 = \mathbb{F}_{p^{12}}$. These parts are discussed in this section and Sect. 4.5, respectively.

Fig. 4(a) illustrates the implementation and timing diagrams of the pairings and $\mathbb{F}_{p^{12}}$ computations of Alg. 1 (i.e., lines 1–6). In this part, the SW domain controls the execution-flow with parallel symmetric or asymmetric executions while all pairings and $\mathbb{F}_{p^{12}}$ operations are computed in the HW domain. First, $2n^2 + 1$ pairings are calculated with the N parallel CP cores through $\lceil (2n^2 + 1)/N \rceil$ iterations. They are shown with blue blocks in Fig. 4(a). In this step, the SW domain sends all inputs (i.e., $g_1^{\mathbf{a}_i}$ and $g_2^{\mathbf{b}_i}$) into the LDMs of the CP cores and the pairing outputs (i.e., e_i) are stored in the GDM or, when the space limit of the GDM is reached, are sent to memories of the SW domain. These communications are shown with orange and gray blocks in Fig. 4(a), respectively. Second, $\mathbb{F}_{p^{12}}$ computations (i.e., exponentiations, inversions, and multiplications) are calculated with the N parallel CP cores. These computations are shown with green blocks in Fig. 4(a). In this step, all inputs are loaded to the LDMs of the CP cores by the GDM of the HW domain and/or SW domain memories (i.e., orange and gray blocks in Fig. 4(a)). Furthermore, several segment and full microprogram packs are transferred from the GPM to all LPMs of the CP cores during each pairing (and $\mathbb{F}_{p^{12}}$ arithmetic operation) in the HW domain. These communications are shown with narrow orange blocks in Fig. 4(a).

4.5 Discrete Logarithms

Alg. 4 describes the parallel baby-step giant-step algorithm for discrete logarithms with N parallel CP cores. Compared to the standard baby-step giant-step algorithm shown in Alg. 3, it introduces two main improvements: (1) the computation is efficiently allocated to N parallel CP cores and (2) it splits the computation to precomputation and on-the-fly computation. Precomputation comprises the baby-step phase of Alg. 3 and makes it a one-time effort needed only in the beginning. On-the-fly computation is the giant-step phase of Alg. 3 and must be computed separately for each input.

Regarding the former improvement, the computations of Alg. 4 are almost optimally distributed to N parallel CP cores resulting in a speedup factor that is very close to N. The latter improvement gives a significant speedup because the baby-step phase typically contributes most of the delay as it always needs to be computed in full whereas the giant-step phase terminates as soon as a match is found. We designed also a Constant-Time (CT) variant for cases where timing attacks are a threat (see Sect. 5.4 for more discussion). The FE-QF decryption guarantees that the result of a decryption from valid ciphertexts is in the interval $[-B, B]$ that is determined by the function f. Consequently, the precomputation phase must compute a table of size $\lceil \sqrt{B} \rceil$. However, in this work, we chose to precompute a table of size B_p where B_p is the maximum size that fits into the memory so that our implementation supports all functions, for which the output bound B satisfies $B_p \geq \lceil \sqrt{B} \rceil$, without the need of new precomputations; obviously, the price of this choice is a slightly longer precomputation phase. Then again, this choice improves the speed of the on-the-fly computation even for functions with smaller output bounds because fewer iterations are needed in the giant-step phase (in particular, the on-the-fly computation terminates before entering the main for loop if the result is at most B_p). The minor improvements of Alg. 4 over Alg. 3 include support for interval $[-B, B]$ where $B \ll \nu$ and includes the negative values. Notice that the last line (i.e., line 18) is never reached with inputs fulfilling all input requirements of Alg. 4, but it is included to cleanly capture decryption failures with faulty ciphertexts. We describe the precomputation and on-the-fly phases with more details in the following subsections.

Algorithm 4: Parallel baby-step giant-step algorithm for discrete logarithms with precomputation using N parallel cores (where N is even). (a) The precomputation phase computes a table T with a memory bound B_p and (b) the on-the-fly computation phase solves the discrete logarithm in the interval $[-B, B]$ given that $B_p \geq \lceil \sqrt{B} \rceil$.

`// (a) Precomputation phase`

Input: $\alpha \in \mathbb{G}$; a function h; a memory bound B_p
Output: Table T, values γ'_u for $1 \leq u \leq N/2$ and δ'

$\mu \leftarrow B_p\,;\ \delta \leftarrow \alpha^N$
Add $(0, h(1 \in \mathbb{G}))$ to a shared table T
for $0 \leq j < \lceil \mu/N \rceil$ **do**
  parallel $1 \leq u \leq N$ **do** `// Assign to` $\mathtt{CP}_1, \ldots, \mathtt{CP}_N$
    if $j = 0$ **then** $\gamma_u \leftarrow \alpha^u$
    else $\gamma_u \leftarrow \gamma_u \cdot \delta$
    Add $(u + j \cdot N, h(\gamma_u))$ to a shared table T
Sort the shared table T in ascending order on column $h(\alpha^j)$ values
$\alpha' \leftarrow \alpha^{-\mu}\,;\ M = N/2\,;\ \delta' \leftarrow \alpha'^M$
parallel $1 \leq u \leq M$ **do** `// Assign to` $\mathtt{CP}_1, \ldots, \mathtt{CP}_{N/2}$
  $\gamma'_u \leftarrow \alpha'^u$
return $T, \gamma'_1, \ldots, \gamma'_M, \delta'$

`// (b) On-the-fly computation phase`

Input: $\beta \in \mathbb{G}$ such that $\beta = \alpha^x$ with $-B \leq x \leq B \in \mathbb{Z}$; a function h; the table T with a memory bound $B_p \geq \lceil \sqrt{B} \rceil$; values γ'_u for $1 \leq u \leq M = N/2$ and δ'
Output: $x \in \{-B, \ldots, B, \perp\}$ where $\perp$ denotes failure

$\mu \leftarrow B_p\,;\ \mu' \leftarrow \lceil B/\mu \rceil - 1\,;\ \beta_{(+)} \leftarrow \beta\,;\ \beta_{(-)} \leftarrow \beta^{-1}$
if $\exists (j, h(\alpha^j)) \in T$ *such that* $h(\alpha^j) = h(\beta_{(+)})$ **then**
  return $x = j$ `// For all` $0 \leq x \leq B_p$
if $\exists (j, h(\alpha^j)) \in T$ *such that* $h(\alpha^j) = h(\beta_{(-)})$ **then**
  return $x = -j$ `// For all` $-B_p \leq x < 0$
for $0 \leq i < \lceil \mu'/M \rceil$ **do**
  parallel do
    parallel $1 \leq u \leq M$ **do** `// Assign to` $\mathtt{CP}_1, \ldots, \mathtt{CP}_{N/2}$
      if $i = 0$ **then** $\gamma'_{u(+)} \leftarrow \gamma'_u \cdot \beta_{(+)}$
      else $\gamma'_{u(+)} \leftarrow \gamma'_{u(+)} \cdot \delta'$
      if $\exists (j, h(\alpha^j)) \in T$ *such that* $h(\alpha^j) = h(\gamma'_{u(+)})$ **then**
        return $x = (u + i \cdot M) \cdot \mu + j$
    parallel $1 \leq u \leq M$ **do** `// Assign to` $\mathtt{CP}_{N/2+1}, \ldots, \mathtt{CP}_N$
      if $i = 0$ **then** $\gamma'_{u(-)} \leftarrow \gamma'_u \cdot \beta_{(-)}$
      else $\gamma'_{u(-)} \leftarrow \gamma'_{u(-)} \cdot \delta'$
      if $\exists (j, h(\alpha^j)) \in T$ *such that* $h(\alpha^j) = h(\gamma'_{u(-)})$ **then**
        return $x = -((u + i \cdot M) \cdot \mu + j)$
return $\perp$

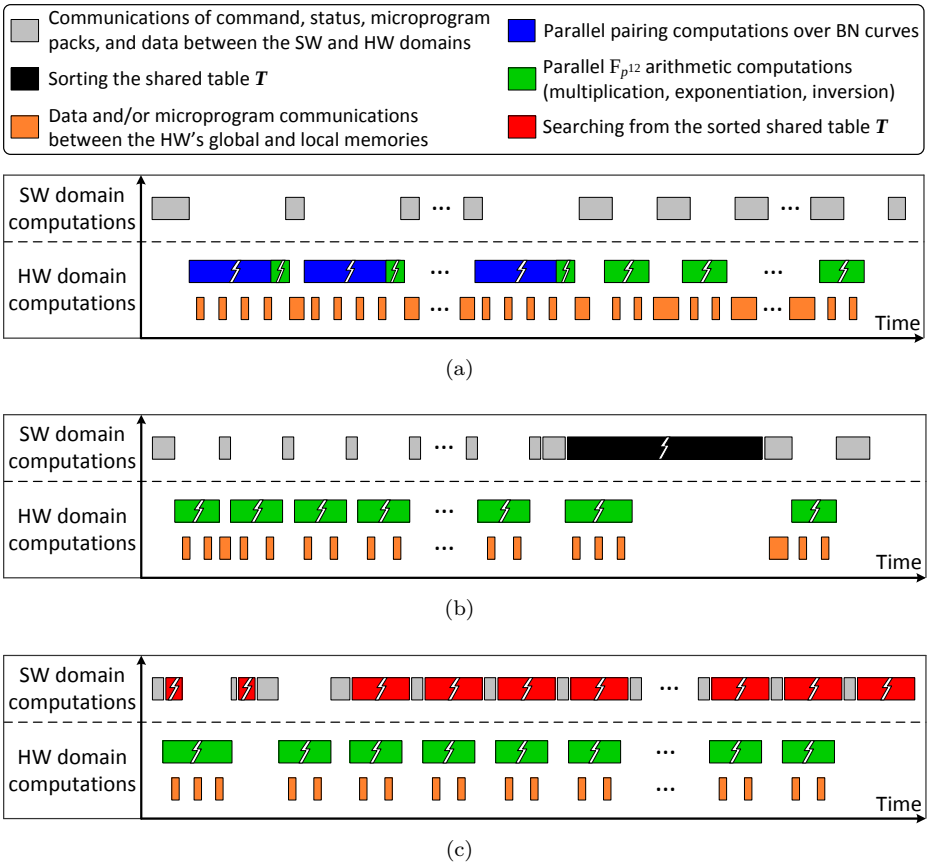

Figure 4: Implementation and timing diagrams of (a) the parallel pairings and $\mathbb{F}_{p^{12}}$ arithmetic computations of Alg. 1 (i.e., lines 1-6), (b) the precomputation part of Alg. 4(a), and (c) the on-the-fly computation part of Alg. 4(b).

4.5.1 Precomputation Phase

As mentioned in Sect. 2.3.2, g_3 in Alg. 1 is a domain parameter (the element $g_3 = e(g_1, g_2) \in \mathbb{F}_{p^{12}}$) that can be fixed. Because the shared table T in Alg. 4 includes powers of $\alpha = g_3$ from 0 to B_p, it can be precomputed. Based on the specifications of our implementation platform, we chose $B_p = 2^{27}$ and to precompute the table when the platform boots up (instead of storing it to non-volatile storage). In Alg. 4, we also employ a function h, which truncates the value of α^j to ℓ bits, to decrease the memory space for storing T and to increase the performance of data communication from the HW domain to the SW domain. We used $\ell = 64$ which still guarantees the uniqueness of each $h(\alpha^j)$.

Fig. 4(b) illustrates the implementation method and timing diagram of Alg. 4(a) in the HW/SW codesign. The precomputation phase contains a main for-loop (i.e., line 3 of Alg. 4(a)), and the SW domain controls and executes its iterations by assigning N parallel $\mathbb{F}_{p^{12}}$ computations (i.e., lines 5 and 6) to the CP cores. These computations are shown with green blocks of Fig. 4(b). In each iteration, all CP cores send the ℓ-bit results (i.e., $h(\gamma_u)$ for $u = 1, 2, \ldots, N$) to the SW domain; this is shown with narrow gray blocks in Fig. 4(b). The SW domain stores the tuples $(j, h(\alpha^j))$ of the shared table T in the DDR memory (see Fig. 3). In addition to the table T, Alg. 4 precomputes also δ' and γ'_u for $u = 1, 2, \ldots, N$. The parallel computations in lines 4–7 compute also α^μ, which can be used to be compute α' through an inversion in line 9. As the figure shows, after computing

lines 1–9, the SW domain initiates the parallel computations of lines 10–11 in the HW domain (i.e., gray, green, and orange blocks). At the same time, it sorts the shared table T (i.e., the black block) using a quicksort routine. A sorted table allows efficient binary searches from the table during the on-the-fly computation phase.

4.5.2 On-The-Fly Computation Phase

This phase is the on-the-fly computation that must be executed separately for each FE-QF decryption. It contains two types of operations: (1) searching from the sorted shared table T and (2) performing $\mathbb{F}_{p^{12}}$ multiplications. They are handled in parallel in the SW and HW domains, respectively. Fig. 4(c) describes the implementation and timing diagram of the on-the-fly computation. At the beginning, the SW domain loads the LDM and LPM of a CP core with the input data (i.e., β) and the microprogram packs of the inversion. It is shown with the first gray block of Fig. 4(c). Then, the same CP core is initiated to compute a $\mathbb{F}_{p^{12}}$ inversion needed for $\beta_{(-)}$ in line 1 of Alg. 4(b), which is shown with the first green block in Fig. 4(c). Simultaneously, the SW domain performs the first search from the table T (i.e., line 2 of Alg. 4(b)) which are shown with the first narrow red block of Fig. 4(c). After computing the $\mathbb{F}_{p^{12}}$ inversion, the SW domain performs the second search from the table T (i.e., line 4 of Alg. 4(b)) which are shown with the second narrow red block of Fig. 4(c). After that, the SW domain loads the LDMs and LPMs of the CP cores with the input data (i.e., γ'_u, and δ') and the microprogram packs of the multiplication. It is shown with the third gray block of Fig. 4(c). Then, the SW domain controls and executes the main for-loop (i.e., line 6 of Alg. 4(b)) by assigning N parallel $\mathbb{F}_{p^{12}}$ multiplications (i.e., lines 9–10 and 14–15) to the CP cores, which are shown with the green blocks in Fig. 4(c). In each iteration, all CP cores send the ℓ-bit results (i.e., $h(\gamma'_u)$ for $u = 1, 2, \ldots, N$) to the SW domain as shown with the narrow gray blocks in Fig. 4(c). In the main for-loop, the SW domain performs N sequential binary searches from the sorted shared table T (i.e., lines 11 and 16) in each iteration. The parallel computations in lines 8–12 and 13–17 correspond to the positive and negative output values, respectively. This process continues until the SW domain finds a match in T and then returns the output x. This is illustrated with the consecutive and parallel red and green blocks in Fig. 4(c).

5 Results and Analysis

In this section, we present and analyze the implementation results. To the best of our knowledge, we presented the first hardware accelerator for an FE-QF scheme and, therefore, we compare our implementation results with the state-of-the-art software implementations.

5.1 Experimental Setup

In order to evaluate the performance of the HW/SW codesign accelerator, we implemented it on real hardware. We targeted Xilinx programmable SoCs and specifically we used the Zynq UltraScale+ MPSoC ZCU102 evaluation kit including a Xilinx Zynq UltraScale+ MPSoC XCZU9EG-2FFVB1156 device, which features a quad-core ARM Cortex-A53 processor running up to 1.5GHz in the SW domain and a 16nm FinFET+ based FPGA in the HW domain. For the SW domain, we used C programming and Xilinx Software Development Kit (SDK) as the development environment. For the HW domain, we used Verilog (HDL) and Xilinx Vivado v2019.1 tool for compiling and implementing the design to the FPGA. The source codes are publicly available[3].

Area Consumption The resource requirements for the HW domain are shown in Table 1. We were able to fit $N = 16$ CP cores into the FPGA by using 33,057 slices (96.5 %), 401

Table 1: Resource requirements of the HW domain (FPGA) in Xilinx Zynq UltraScale+ MPSoC ZU9EG device. The clock frequency constraint was set to 210 MHz in Vivado.

Component	LUTs	Flip-Flops	CLB Slices†	DSPs	BRAM Tiles
Local interface unit	902	79	161	0	2
LDM unit	67	0	31	0	16
LPM unit	716	42	138	0	4
Control unit	184	67	37	0	0
Datapath unit	7,276	10,640	1,602	36	0
Single CP core	9,145	10,828	1,969	36	22
(% of overall FPGA)	3.3%	2.0%	5.7%	1.9%	2.4%
Global interface unit	1,925	164	339	0	4
GDM unit	897	0	228	0	8
GPM unit	74	0	46	0	32
Parallel CPs ($N = 16$)	146,320	173,248	30,205	576	352
Other blocks‡	5,259	9,027	2,239	0	5
Multi-core HW domain	154,475	182,439	33,057	576	401
(% of overall FPGA)	56.4%	33.3%	96.5%	22.8%	44.0%

† Each CLB slice contains eight 6-input LUTs and sixteen flip-flops.
‡ This unit contains AXI connections (DMA, peripheral and memory interconnect blocks, and several GPIO blocks), command and status logic, and processor system reset blocks.

BRAMs (44.0 %), and 576 DSPs (22.8 %). The maximum clock frequencies for the FPGA (i.e., HW domain) and ARM (i.e., SW domain) are 210 and 1,200 MHz, respectively[5]. Based on Vivado, the total on-chip power consumption is about 17.47 W. All reported results are final post-place&route results and validated with real hardware.

Timing results of the SW/HW primitive operations Table 2 presents total execution times and HW latencies of different low-level operations. It includes four parts. (1) The top part presents the initialization and precomputation part. The initialization step initializes all SW and HW memories with data and microprograms and configures all HW domain modules. The precomputation computes the sorted shared table T for the discrete logarithms using Alg. 4(a). In this work, we chose the largest memory bound, for which the precomputed table T still fits into the DDR memory of the evaluation kit. Specifically, we have $B_p = 2^{27}$ that supports up to 54-bit outputs and results in a 1.9 GB precomputed table T. It takes 535 seconds to perform all operations required to construct a sorted table T. Although this part needs to be executed only once at boot-up, it takes long and can be problematic in some settings. In such cases, the precomputed table T could be stored in a non-volatile memory (e.g., an SD card) or loaded via an external interface at boot-up. (2) The second part shows the timings for interacting different command/status, data, and microprogram packs between different domains and local/global memories. (3) The third part shows the timings for computing $\mathbb{F}_{p^{12}}$ arithmetic. Squaring, multiplication, and inversion are CT by default, but exponentiation has two versions: a Variable-Time (VT) square-and-multiply and CT square-and-multiply-always. (4) The bottom part reports the timing for an optimal ate pairing calculation, which is CT.

[5]The maximum clock frequency for a single instance of the CP core is 230 MHz [BJ20a] so the frequency drop to 210 MHz with $N = 16$ is only a minor one and greatly outweighed by the advantages of parallelization. Arguably, the reason why the high slice utilization (96.5 %) did not cause a more significant frequency drop is the fact that the slices are relatively sparsely filled as the LUT utilization is only 56.4 %.

Table 2: Total execution times and HW latencies of different SW/HW primitives.

SW/HW primitive operation	HW cycles	Total time (μs)
Init., config., and program loading of HW units	27,103	267.946
Precomputation phase for $B_p = 2^{27}$ (Alg. 4(a))	5,712,944,669	535,450,051.424
32-bit command/status transfer SW$\leftrightarrow$HW	5/4	0.047/0.042
256-bit word transfer SW$\leftrightarrow$GDM/LDM	9/11	0.072/0.081
256-bit word transfer GDM$\leftrightarrow$LDM	5	0.024
Segment/full pack[†] transfer SW$\rightarrow$GPM	26/584	0.339/11.156
Segment/full pack transfer SW$\rightarrow$LPM	28/586	0.348/11.166
Segment/full pack transfer GPM$\rightarrow$LPM	12/291	0.052/1.386
Squaring/multiplication in $\mathbb{F}_{p^{12}}$ (CT)	475/703	2.262/3.348
Exponentiation in $\mathbb{F}_{p^{12}}$ (Exp-CT/Exp-VT)[‡]	12,028/9,223	57.276/43.919
Inversion in $\mathbb{F}_p/\mathbb{F}_{p^{12}}$ (CT)	11,938/13,511	56.848/64.338
Optimal ate pairing (Alg. 2)	200,158	982.597

[†] Segment and full pack sizes are 2.25Kb (32×72b) and 72Kb (1024×72b), resp.
[‡] Exp-CT/VT assume a 11-bit exponent; Exp-VT also assumes Hamming weight of 6.

5.2 Implementation Results of the FE-QF Algorithms

Fig. 5(a) depicts the timing results for the pairings and $\mathbb{F}_{p^{12}}$ arithmetic part of Alg. 1 (i.e., lines 1–6) as a function of n, the length of $\mathbf{x}$ and $\mathbf{y}$, for both general and diagonal f. As discussed in Sect. 2.3, a diagonal f requires $2n + 1$ pairings (and arithmetic in $\mathbb{F}_{p^{12}}$) whereas a general f requires $2n^2 + 1$ pairings. This difference is clearly visible in Fig. 5(a) which shows that, when $n = 140$, a diagonal f requires only about 22 ms compared to about 3 s required by a general f. Fig. 5(b) presents the timing results for the discrete logarithm part of Alg. 1 (i.e., line 7) as a function of the bit-length of the output result in bits. Because we use a precomputed table with $B_p = 2^{27}$, the computation for at most 27-bit results takes around 0.11 ms because the result is directly available in T. For larger outputs, the computation time increases quickly. For example, for output sizes of 29, 38, and 54 bits, the computation times are 1.2 ms, 17.5 ms, and 1,096,478 ms, respectively. Hence, functions, for which results are typically small, are relatively fast to compute with the VT discrete logarithm algorithm even if the theoretical bound B_t for that function is large. The CT-variant of the discrete logarithm computation always executes all iterations of Alg. 4 up to μ' required by the maximum output bound B_t of that particular function f. Hence, the cost of CT vs. VT discrete logarithms depends on both B_t and the distribution of the outputs for the particular function and use case. Fig. 5 allows estimating the total execution time of a FE-QF decryption with Alg. 1 for different functions and features.

Table 3 collects timing results for functions of different sizes and types (i.e., general and diagonal f) and for both VT and CT variants. The CT variant computes discrete logarithms always with the maximum number of iterations for that function whereas the VT variant terminates immediately after the result is found. The VT timings assume uniformly random input vectors $\mathbf{x}$, $\mathbf{y}$, and function f. Table 3 presents also the timing results for the worst cases of the n and B_t parameters of the two use cases from Sect. 6. As can be seen, the difference between CT and VT results grows when B_t increases for both general and diagonal f. This trend is more obvious for diagonal f because the discrete logarithm computation is more dominant thanks to fewer number of pairings.

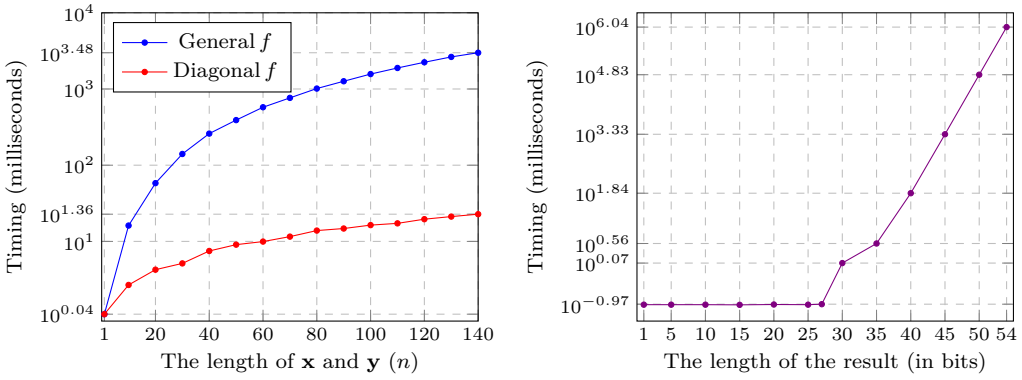

Figure 5: Timing results of the HW/SW codesign for (a) pairings and $\mathbb{F}_{p^{12}}$ arithmetic (Alg. 1 lines 1–6) as a function of n, the length of $\mathbf{x}$ and $\mathbf{y}$, and (b) discrete logarithm (Alg. 1 line 7) as a function of the length of the result in bits.

Table 3: Timing results for functions with different parameters n and B_t (in milliseconds). The VT times are calculated with uniformly random input vectors x, y and function f.

FE-QF decryption variant	Times in ms for (n , B_t in bits)				
	(10 , 20)	(20 , 27)	(30 , 30)	(40 , 33)†	(128 , 41)‡
CT, general f	16.244	58.224	141.516	261.916	2,731.384
CT, diagonal f	2.786	4.361	6.315	9.606	156.002
VT, general f	15.757	56.334	136.199	253.409	2,535.900
VT, diagonal f	2.678	4.170	5.084	8.281	36.757

† The worst case in MNIST dataset.
‡ The worst case in Fashion MNIST dataset.

5.3 Comparison and Discussion

Table 4 presents a performance comparison against two software implementations. The decryption timings are for the general f with the maximum output bound (i.e., $B_t = n^2 \cdot B_x \cdot B_y \cdot B_f$). The first software results (the original GoFE library) are from [MSH+19] and contain timings of key generation, encryption, and decryption with different parameters. It is evident that it becomes impractical for larger parameters n and B_t due to the slowness of the decryption. Moreover, it can be observed that key generation and encryption are considerably faster and do not form significant performance challenges for the real-life FE-QF schemes, supporting our choice to focus on decryption in this work. The original GoFE library reported in [MSH+19] does not use precomputation tables for discrete logarithms. Hence, we included the second software results from an optimized GoFE library that uses Alg. 4 with $B_p = 2^{25}$ for discrete logarithms. This significantly improved the software decryption timings and provides fairer comparisons against software.

Table 4 shows that our HW/SW codesign offers speedups between 2.3–20.0 times compared to the optimized GoFE library and up to over 1000 times compared to the original GoFE library from [MSH+19]. The results demonstrate that the original GoFE library suffers from slow discrete logarithms and, hence, the use of Alg. 4 provides major improvements also in software. Compared to the optimized GoFE library, the largest speedups are obtained when the number of pairings is high.

Table 4: Performance comparison of the FE-QF algorithms against software implementations (in seconds). Decryption timing results are for uniformly random x, y, f (with inputs from $[-B_{x,y,f}, B_{x,y,f}]$) and decryption upper bound $B_t = n^2 \cdot B_x \cdot B_y \cdot B_f$.

Parameters			GoFE [MSH+19]			GoFE †	**HW/SW codesign**	
$B_{x,y,f}$	n	B_t	KeyGen	Enc.	Dec.	Dec.	Dec.	Speedup ‡
100	1	20-bit	0.0001	0.0241	0.0910	0.0050	0.0012	75.8 / 4.2
100	5	23-bit	0.0001	0.1104	0.7502	0.0760	0.0051	147.1 / 14.9
100	20	29-bit	0.0002	0.4334	9.8077	1.1810	0.0591	166.0 / 20.0
500	1	27-bit	0.0001	0.0251	0.4986	0.0050	0.0012	41.6 / 4.2
1000	1	30-bit	0.0001	0.0251	1.1859	0.0050	0.0022	539.0 / 2.3
1000	20	39-bit	0.0002	0.4541	92.3084	1.4760	0.0805	1146.7 / 18.3

† Optimized GoFE library using Alg. 4 with $B_p = 2^{25}$.
‡ Factors compared to both GoFE versions.

5.4 Discussion on Side-Channels

Side-channel attacks are a significant threat to practical cryptosystems and implementations should include countermeasures against attacks that are considered possible in the threat models of applications where the implementations could be deployed. FE has certain features that provide inherent resilience against side-channel attacks. E.g., if we consider the specific case of our HW/SW codesign for FE-QF decryptions, the only (potentially) sensitive values handled by the implementation are the decryption keys dk_f and the results of the decryptions. By the nature of FE, the decryption keys do not provide full access to the plaintexts reducing their lucrativeness as targets of side-channel attacks compared to keys of traditional cryptosystems. Nevertheless, one may envision applications where they need to be protected against side-channel attacks. The output results may also be sensitive and an attacker could try to learn information about them via side-channel attacks.

We consider timing attacks as the main threats for our implementation because, in most envisioned applications, the implementation would be installed as an accelerator for a server and it is unlikely that attackers could get physical access to the device. Hence, we have considered CT implementation as the main method to protect against side-channel attacks. Notice that—in our case—this implies protection even against simple power and electromagnetic attacks because the operation patterns are constant. We leave protections against advanced power attacks as topics for future research.

The decryption key $\mathsf{dk}_f = \left(g_2^{f(\mathbf{s},\mathbf{t})}, f\right)$ is used in Alg. 1 in line 1 in a pairing computation and in line 6 as an exponent in exponentiations in $\mathbb{F}_{p^{12}}$. The pairings in our implementation are CT by default so line 1 does not leak any information about $g_2^{f(\mathbf{s},\mathbf{t})}$ via the timing channel. In the VT variants, the computations of line 6 leak information about the function f, but not about the entire decryption key dk_f. I.e., even if the attacker learns f, (s)he cannot decrypt any ciphertexts with that information alone as $g_2^{f(\mathbf{s},\mathbf{t})}$ would still be required. If even the function itself is secret, then it could be protected against timing attacks by adopting a CT exponentiation algorithm. The CT variants in Table 3 utilize CT square-and-multiply-always exponentiations. Notice that there are also function hiding FE schemes (e.g., [BRS13, BJK15, ACF+18]), which would hide f even from a legitimate decryptor but they are out of the scope of this work.

The discrete logarithm in line 7 of Alg. 1 does not utilize the decryption key but may still require side-channel protections if the output result is sensitive. In particular, the timing of the computation with Alg. 4(b) relies heavily on the size of the result. E.g., if the result is smaller than B_p, the computation terminates almost immediately (0.11 ms), but if the result is close to the maximum value 2^{54}, the computation takes several minutes.

Consequently, an outside observer could learn information about decryption results simply by measuring response times. However, in a normal setting, the variation of discrete logarithm timings is much more moderate than in the above extreme case (see, e.g., Fig. 7 for output bit-length distributions in the two use cases). Anyways, this leakage can be prevented by using the CT variant of the discrete logarithm algorithm where all iterations of the for loop in lines 6–17 of Alg. 4(b) are always computed regardless of when the match is found. It is crucial that the number of iterations is determined by using the bounds of the particular function f and not by using the maximum bound supported by the implementation (2^{54}). Table 3 shows that the cost of CT discrete logarithms starts to dominate when the upper bound for the output is large. E.g., with at most 33-bit results the overhead compared to the VT variant is only 16 % with uniformly distributed inputs, but it is already 324 % for at most 41-bit results.

To conclude, the implementation can be fully protected against timing attacks, but the cost depends heavily on the function that is computed. Hence, the use of the countermeasures should be decided based on the performance and threat model of the application.

6 Use Cases

Design and implementation of ML models on the encrypted data are difficult and require special techniques like FE. The performance of such models is typically not comparable to the performance of the models that are executed on the unencrypted data. In what follows, we show how the performance of such models can be significantly improved by using a hardware accelerator.

6.1 ML Classification on the Encrypted MNIST Dataset

One of the main tasks that demonstrate the power of ML is image classification. Contemporary ML algorithms are able to solve tasks such as recognition of traffic signs or deciding a patient's condition based on his/her medical image. In many cases images contain private information that should be kept secret. Using traditional encryption, nothing can (and must not) be learned from the encrypted data. FE, on the other hand, permits image classifications without revealing the images themselves.

A basic example used to test ML algorithms is the MNIST (Modified National Institute of Standards and Technology) dataset. It consists of 60,000 images of handwritten digits (28×28 pixels), and a test set of 10,000 images. The task is to classify the images into 10 classes based on which digit is written in the image. Modern ML algorithms can achieve above 99.7% accuracy in this task. However, practical FE supports only limited functions that can be evaluated and such accuracy cannot be reached.

Papers [LCFS17, DGP18, MSH+19] demonstrated how the classification of handwritten digits can be done using FE-IP and FE-QF, where the latter is achieving higher accuracy. The proposed model is a neural network with one hidden layer where the non-linear function used as the activation function in the hidden layer is the element-wise quadratic function. The output layer is of dimension 10 and can be interpreted as the likelihoods of each 10 digits being in the image. A different dk_{f_i} is provided for each digit i and the decryption that provides the largest output value is interpreted as the digit in the image. The FE-QF scheme in [DGP18] has homomorphic properties that can be used as a linear transformation between the input and hidden layers. Moreover, the evaluation of 10 quadratic functions with diagonal entries can be used to evaluate the activation function and the linear transformation to the output layer. Such a model—which is in fact equivalent to [DGP18]—achieves an accuracy of about 97% [DGP18, MSH+19]. We report the timings for FE-QF decryptions from images that have already been projected to support decryption with diagonal f. Fig. 6 gives a visual representation of this use case.

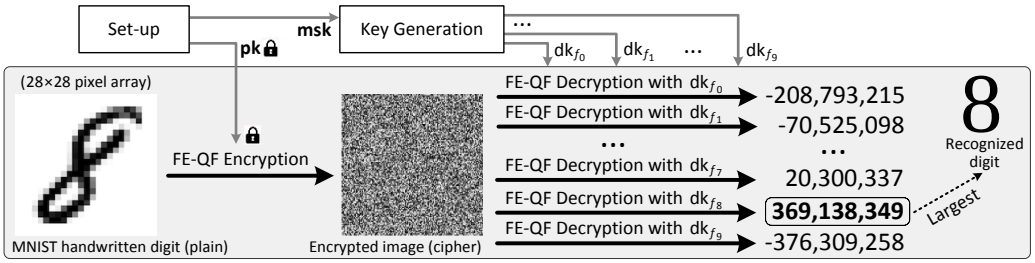

Figure 6: Visual representation of the MNIST use case.

Training of the model needs to be done on unencrypted data, while classification is done on encrypted images. The images have been presented as 785-coordinate vectors ($28 \cdot 28$ and one for bias) and we use $n = 40$ dimensional hidden layer. The inputs as well as linear transformations between layers must be discretized to be used by FE. Discretization to greater intervals of integers leads to slow decryption, while discretization to small intervals decreases the prediction accuracy. We chose $B_x = 4$ for the inputs and $B_1 = 200, B_2 = 40$ as the discretization bounds for the first and the second linear transformation. Choosing such bounds, the accuracy does not significantly change and the output values can be bounded with $B_t = 2^{39}$; however, they are much smaller in practice (see Fig. 7).

6.2 ML Classification on the Encrypted Fashion MNIST Dataset

The prediction power of an ML model used with FE depends on both the ML techniques (e.g., model design, training algorithm, etc.) and the functionality and performance of FE. In this paper we demonstrated how to boost the performance of FE-QF using HW/SW codesign. A speedup in the FE part implies that the model can use more parameters and a less strict discretization and, consequently, achieve better accuracy while remaining practical. In the neural network model, the hidden layer can have a higher dimension and the discretization for inputs and linear transformations can use wider ranges.

Fashion MNIST is a dataset that is very similar to MNIST but consists of images of 10 different clothing objects (e.g., shoes, t-shirts, coats, etc.). The task of classifying clothes is harder than classifying digits and the most advanced algorithms achieve about 95% accuracy on Fashion MNIST (compared to 99.7% on MNIST). Using the parameters of the MNIST model on Fashion MNIST dataset gives poor classification accuracy. However, increasing the size of the hidden layer to $n = 128$ and the discretization bounds to $B_x = 10$, $B_1 = 1000$, and $B_2 = 50$ achieves an accuracy of above 87.5%. Consequently, the number of pairings is higher and the output values are larger (see Fig. 7) leading to slower discrete logarithms. However, these numbers are still manageable by our HW/SW design.

6.3 Results and Comparisons

Table 5 presents the result for the ML classification models on encrypted MNIST and Fashion MNIST images using our accelerator. Moreover, we compare the performance with both original and optimized GoFE libraries. Table 5 shows that the HW/SW design gives a similar speedup as reported in the previous section even for real use cases.

7 Conclusions

In this paper, we presented the first published accelerator architecture for FE-QF. We showed that HW/SW-codesign based acceleration results in significant speedups compared to software-only solutions. Moreover, we demonstrated that large speedups can be received

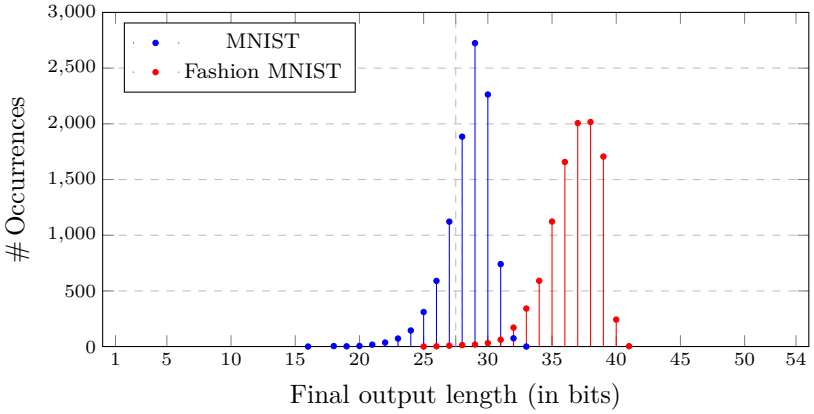

Figure 7: Distributions of the output bit-lengths for the MNIST and Fashion MNIST datasets. The diagrams are plotted from 10,000 images. The mean lengths are 29 and 37 bits for MNIST and Fashion MNIST, respectively. Discrete logarithms to the left of the vertical dashed line are found directly in the precomputed table with $B_p = 2^{27}$.

Table 5: Performance comparison for the ML classification on the encrypted MNIST and Fashion MNIST datasets. Timings (in seconds) include all 10 FE-QF decryptions.

Implementation	Platform	MNIST	Fashion MNIST
Original GoFE [MSH+19]	Core i7 @ 3.40 GHz	< 20 s	—
Optimized GoFE with $B_p = 2^{25}$	Core i7 @ 2.80 GHz	1.344 s	5.221 s
This work	Zynq UltraScale+	0.0866 s	0.3796 s
(Speedup factor)	MPSoC ZU9EG	($\approx 15.5\times$)	($\approx 13.8\times$)

also in real use cases, namely, for image classifications using encrypted MNIST and Fashion MNIST datasets. We anticipate that our results will help FE to become more feasible for practical adaptation, as the overheads of FE are often excessive and have presented obstacles for practical use of FE in real applications. In general, our accelerator allows to pair FE with complex systems, such as FE-QF based on cryptographic pairings with advanced ML models.

We implemented the FE-QF scheme from [DGP18] but also other published FE-QF schemes (e.g., [BCFG17, Gay20, Wee20]) have very similar decryption routines consisting of pairings and discrete logarithms. Consequently, our implementation could be relatively easily adapted to such schemes, too. The discrete logarithm algorithm and its implementation can be used also for other cryptographic schemes including certain FE-IP schemes (e.g., [ABDP15, ALS16, ACF+18]). Such adaptations are topics for future research.

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780108 (FENTEC).

References

[ABC+08] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi.

Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *Journal of Cryptology*, 21(3):350–391, July 2008.

[ABDP15] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 733–751. Springer, Heidelberg, March / April 2015.

[ACF+18] Michel Abdalla, Dario Catalano, Dario Fiore, Romain Gay, and Bogdan Ursu. Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 597–627. Springer, Heidelberg, August 2018.

[AGRW17] Michel Abdalla, Romain Gay, Mariana Raykova, and Hoeteck Wee. Multi-input inner-product functional encryption from pairings. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 601–626. Springer, Heidelberg, April / May 2017.

[ALS16] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 333–362. Springer, Heidelberg, August 2016.

[BCFG17] Carmen Elisabetta Zaira Baltico, Dario Catalano, Dario Fiore, and Romain Gay. Practical functional encryption for quadratic functions with applications to predicate encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 67–98. Springer, Heidelberg, August 2017.

[BDM+10] Jean-Luc Beuchat, Jorge Enrique González Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. Cryptology ePrint Archive, Report 2010/354, 2010. https://eprint.iacr.org/2010/354.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 213–229. Springer, Heidelberg, August 2001.

[BGM+10] Jean-Luc Beuchat, Jorge E. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal Ate pairing over Barreto-Naehrig curves. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *PAIRING 2010*, volume 6487 of *LNCS*, pages 21–39. Springer, Heidelberg, December 2010.

[BJ20a] Milad Bahadori and Kimmo Järvinen. Compact and programmable yet high-performance soc architecture for cryptographic pairings. In Nele Mentens, Leonel Sousa, Pedro Trancoso, Miquel Pericàs, and Ioannis Sourdis, editors, *30th International Conference on Field-Programmable Logic and Applications, FPL 2020, Gothenburg, Sweden, August 31 - September 4, 2020*, pages 176–184. IEEE, 2020.

[BJ20b] Milad Bahadori and Kimmo Järvinen. A programmable SoC-based accelerator for privacy-enhancing technologies and functional encryption. *IEEE Trans. Very Large Scale Integr. Syst.*, 28(10):2182–2195, 2020.

[BJK15] Allison Bishop, Abhishek Jain, and Lucas Kowalczyk. Function-hiding inner product encryption. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 470–491. Springer, Heidelberg, November / December 2015.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 319–331. Springer, Heidelberg, August 2006.

[BRS13] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 461–478. Springer, Heidelberg, August 2013.

[BSW07] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy*, pages 321–334. IEEE Computer Society Press, May 2007.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Heidelberg, March 2011.

[DGP18] Edouard Dufour Sans, Romain Gay, and David Pointcheval. Reading in the dark: Classifying encrypted digits with functional encryption. Cryptology ePrint Archive, Report 2018/206, 2018. https://eprint.iacr.org/2018/206.

[Gay20] Romain Gay. A new paradigm for public-key functional encryption for degree-2 polynomials. Cryptology ePrint Archive, Report 2020/093, 2020. https://eprint.iacr.org/2020/093.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 89–98. ACM Press, October / November 2006. Available as Cryptology ePrint Archive Report 2006/309.

[Jou04] Antoine Joux. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, September 2004.

[KB16] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 543–571. Springer, Heidelberg, August 2016.

[LCFS17] Damien Ligier, Sergiu Carpov, Caroline Fontaine, and Renaud Sirdey. Privacy preserving data classification using inner-product functional encryption. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017*, pages 423–430. SciTePress, 2017.

[MOV93] Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Inf. Theory*, 39(5):1639–1646, 1993.

[MSH+19] Tilen Marc, Miha Stopar, Jan Hartman, Manca Bizjak, and Jolanda Modic. Privacy-enhanced machine learning with functional encryption. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019, Part I*, volume 11735 of *LNCS*, pages 3–21. Springer, Heidelberg, September 2019.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010. https://eprint.iacr.org/2010/556.

[SBC+09] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In Hovav Shacham and Brent Waters, editors, *PAIRING 2009*, volume 5671 of *LNCS*, pages 78–88. Springer, Heidelberg, August 2009.

[SDK17] Ahmad Salman, William Diehl, and Jens-Peter Kaps. A light-weight hardware/software co-design for pairing-based cryptography with low power and energy consumption. In *International Conference on Field Programmable Technology, FPT 2017, Melbourne, Australia, December 11-13, 2017*, pages 235–238. IEEE, 2017.

[SGZ+15] Anissa Sghaier, Loubna Ghammam, Medyen Zeghid, Sylvain Duquesne, and Mohsen Machhout. Area-efficient hardware implementation of the optimal ate pairing over BN curves. Cryptology ePrint Archive, Report 2015/1100, 2015. https://eprint.iacr.org/2015/1100.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proceedings of Symposium in Pure Mathematics*, volume 20, pages 415–440. American Mathematical Society, 1971.

[Ver10] Frederik Vercauteren. Optimal pairings. *IEEE Trans. Inf. Theory*, 56(1):455–461, 2010.

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 678–697. Springer, Heidelberg, August 2015.

[Wee20] Hoeteck Wee. Functional encryption for quadratic functions from k-lin, revisited. In *TCC 2020, Part I*, LNCS, pages 210–228. Springer, Heidelberg, March 2020.

[YFCV13] Gavin Xiaoxu Yao, Junfeng Fan, Ray C. C. Cheung, and Ingrid Verbauwhede. Faster pairing coprocessor architecture. In Michel Abdalla and Tanja Lange, editors, *PAIRING 2012*, volume 7708 of *LNCS*, pages 160–176. Springer, Heidelberg, May 2013.

A Adaptability to Other SoC Architectures

In this appendix, we provide a brief discussion on the efforts and modifications that would be required to use the proposed SoC architecture (i.e., HW/SW codesign) in a different SoC platform than Xilinx Zynq UltraScale+ MPSoC. The proposed HW/SW codesign architecture is generic and only minor modifications are needed to instantiate it in various

programmable SoCs, including other FPGA vendors (e.g., Intel) and soft-core CPUs (e.g., RISC V, NIOS II, Microblaze). In the current HW/SW codesign system instantiated on the Xilinx Zynq UltraScale+ MPSoC, we applied a 128-bit high-performance interface (i.e., AXI HP) and several 32-bit general-purpose (i.e., AXI GP) interfaces between the SW and HW domains for communicating data/microprogram and command/status packets, respectively. Therefore, the main modifications would be required in the HW/SW interfaces because HDL codes for the HW domain (i.e., Verilog source codes) and C codes for the SW domain are mostly generic. More precisely, the current implementation uses 128-bit AXI HP and 32-bit AXI GP interfaces (of Xilinx SoC platforms), but other SoCs may require different interfaces with the same or different sizes. It should be considered that in the targeted SoC platform, the HW side (i.e., FPGA) contains enough logic cells and block memories for implementing the parallel CP cores (i.e., in this work, 16 CP cores) and global/local memories, respectively. Moreover, the SW domain must be interfaced with an external RAM memory (i.e., off/on-chip DDR memory) with at least 2GB space. If enough resources are not available, then fewer parallel CP cores must be used or the size of the precomputed table for discrete logarithms must be decreased, which may have significant effects on the performance compared to the numbers reported in this paper. Naturally, also the overall performance of the FPGA and the processor cores of the target SoC affect the performance.

B Implementation Details

Stucture of the Arithmetic Unit [BJ20a] Fig. 8(a) and 8(b) describe the structure of the MMMB for computing $\mathbb{F}_p$ multiplications/squarings, and also Fig. 8(c) depicts the structure of the MASB for computing $\mathbb{F}_p$ additions/subtractions. The MMMB contains three nested parts which are organized bottom-up as a Multiply-Add-Add Block (MAAB), a Multiply-Add-Add-Accumulator Block (MAAAB), and the overall structure of the MMMB. The MAAB is the primary computation block in the datapath and consists of a 64×64-bit Karatsuba multiplier (constructed from three parallel 32×32-bit multipliers) combined with adders to compute $a \times b + c + d$ (all 64-bit values) in a five-stage pipeline. The MAAB consumes most of the FPGA resources, has the highest dynamic power consumption, and also contains the critical path of the CP core. In order to maximize its efficiency, it is implemented using the DSP slices. In the next part, the MAAB is complemented with an accumulation operation (i.e., the MAAAB). The lower part of the MAAB result is accumulated with the previous higher part as well as with the previous most significant bit of the accumulation result (i.e., the input carry). The output carry and the higher part of the MAAB result are stored for the next accumulation (see Fig. 8)(b). The latencies for computing r_{low} and r_{high} are five and six clock cycles, respectively. This accumulation method and the one clock cycle difference between r_{low} and r_{high} are essential for efficient implementation of high-radix Montgomery modular multiplication algorithm. Finally, in the top part, MAAAB (as the main computing core) as well as multiplexers, registers, and FSMs are used for implementing radix-2^{64} Montgomery modular multiplication. The MMMB computes a multiplication/squaring in $\mathbb{F}_p$ with a total latency of 43 clock cycles, but a new multiplication/squaring can be started already after 38 clock cycles due to the pipelined scheme. Furthermore, the structure of MASB with a two-stage pipeline is illustrated in Fig.8(c). Addition and subtraction in $\mathbb{F}_p$ can be realized by two consecutive adder/subtractor circuits which produce the result in two cycles. Due to the pipeline, its throughput is one $\mathbb{F}_p$ addition/subtraction per cycle. Applying two MASBs and connecting the outputs of the datapath back to its inputs facilitates efficient field arithmetic operations such as $\mathbb{F}_{p^2}$ addition/subtraction/negation, $\mathbb{F}_{p^2}$ multiplication/squaring, and multiplications by small constants.

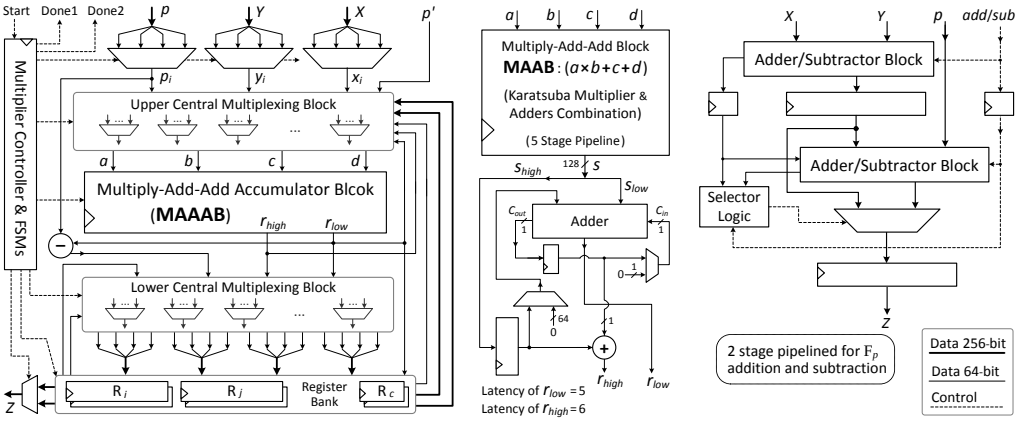

Figure 8: Structures of (a) MMMB, (b) MAAAB, and (c) MASB blocks [BJ20a].

Scheduling of the Tower Field Arithmetic [BJ20a] The timing diagram and scheduling technique of the tower extension field arithmetic are depicted in Fig. 9. On the top, it shows how three parallel $\mathbb{F}_p$ multiplications/squarings and several parallel additions/subtractions can be computed simultaneously in the datapath by utilizing parallelism and pipelining. In the middle, it depicts how this scheduling allows computing $\mathbb{F}_{p^2}$ multiplications/squarings effectively in 38 clock cycles and, also, that up to eleven $\mathbb{F}_{p^2}$ additions/subtractions can be done during each $\mathbb{F}_{p^2}$ multiplication/squaring. The $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ operations are further used for implementing $\mathbb{F}_{p^4}$, $\mathbb{F}_{p^6}$, and $\mathbb{F}_{p^{12}}$ arithmetic. An inversion in $\mathbb{F}_{p^{12}}$ is decomposed into several additions/subtractions, multiplications/squarings, and a single inversion in $\mathbb{F}_p$. We compute the inversion in $\mathbb{F}_p$ with Fermat's Little Theorem through the modular exponentiation. We compute this $\mathbb{F}_p$ exponentiation and also $\mathbb{F}_{p^{12}}$ exponentiation using the right-to-left square-and-multiply algorithm because it allows computing multiplications in parallel with squarings and results in more efficient implementation.

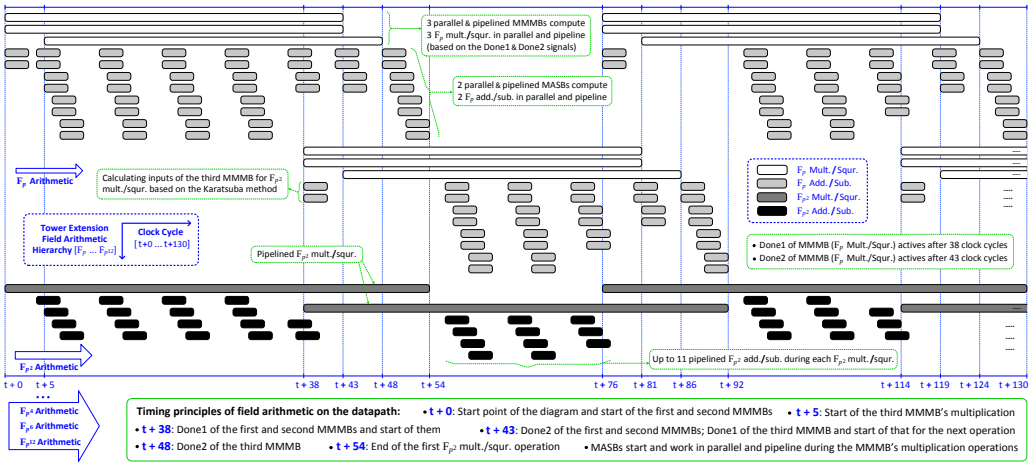

Figure 9: Timing diagram and scheduling technique of the tower extension field arithmetic operations (i.e., $\mathbb{F}_p$, $\mathbb{F}_{p^2}$, $\mathbb{F}_{p^4}$, $\mathbb{F}_{p^6}$, and $\mathbb{F}_{p^{12}}$ arithmetic) in our datapath [BJ20a].