Finesse: An Agile Design Framework for Pairing-based Cryptography via Software/Hardware Co-Design

Tianwei Pan[*]
Beihang University
Beijing, China
pantw@buaa.edu.cn

Tianao Dai[*]
Beihang University
Beijing, China
daitianao@buaa.edu.cn

Jianlei Yang[†]
Beihang University
Beijing, China
jianlei@buaa.edu.cn

Hongbin Jing
Beihang University
Beijing, China
hongbin@buaa.edu.cn

Yang Su
Beihang University
Beijing, China
buaayangsu@buaa.edu.cn

Zeyu Hao
Beihang University
Beijing, China
withinlover@buaa.edu.cn

Xiaotao Jia
Beihang University
Beijing, China
jiaxt@buaa.edu.cn

Chunming Hu
Beihang University
Beijing, China
hucm@buaa.edu.cn

Weisheng Zhao
Beihang University
Beijing, China
weisheng.zhao@buaa.edu.cn

Abstract

Pairing-based cryptography (PBC) is crucial in modern cryptographic applications. With the rapid advancement of adversarial research and the growing diversity of application requirements, PBC accelerators need regular updates in algorithms, parameter configurations, and hardware design. However, traditional design methodologies face significant challenges, including prolonged design cycles, difficulties in balancing performance and flexibility, and insufficient support for potential architectural exploration.

To address these challenges, we introduce `Finesse`, an agile design framework based on co-design methodology. `Finesse` leverages a co-optimization cycle driven by a specialized compiler and a multi-granularity hardware simulator, enabling both optimized performance metrics and effective design space exploration. Furthermore, `Finesse` adopts a modular design flow to significantly shorten design cycles, while its versatile abstraction ensures flexibility across various curve families and hardware architectures.

`Finesse` offers flexibility, efficiency, and rapid prototyping, comparing with previous frameworks. With compilation times reduced to minutes, `Finesse` enables faster iteration cycles and streamlined hardware-software co-design. Experiments on popular curves demonstrate its effectiveness, achieving 34× improvement in throughput and 6.2× increase in area efficiency compared to previous flexible frameworks, while outperforming state-of-the-art non-flexible ASIC designs with a 3× gain in throughput and 3.2× improvement in area efficiency.

CCS Concepts

• **Hardware → Hardware-software codesign**; *Hardware accelerators*; • **Security and privacy → Cryptography**.

Keywords

Agile design framework, pairing-based cryptography, hardware accelerator, software/hardware co-design

ACM Reference Format:
Tianwei Pan, Tianao Dai, Jianlei Yang, Hongbin Jing, Yang Su, Zeyu Hao, Xiaotao Jia, Chunming Hu, and Weisheng Zhao. 2025. Finesse: An Agile Design Framework for Pairing-based Cryptography via Software/Hardware Co-Design. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3695053.3731033

1 Introduction

Bilinear pairing, since its formulation in modern cryptography, has been a crucial primitive for building advanced cryptographic protocols and systems. Pairings enable efficient schemes for identity-based encryption [1], attribute-based encryption [2], short signature [3], SNARKs such as KZG [4] and Groth16 [5]. As data security and privacy gain increasing attention, pairing-based cryptography plays a significant role in this context, safeguarding user data and underpinning the trust and reliability of modern digital infrastructures. Despite its benefits, pairing comes with a significant computational cost. While traditional signature schemes offer a latency of around 20 µs on desktop CPUs, pairing computations are typically 2 orders of magnitude longer [6]. Application needs for pairing have motivated researchers to embark on the exploration of efficient pairing acceleration techniques, targeting platforms

[*] Both authors contributed equally to this research.
[†] Corresponding author is Jianlei Yang. Email: jianlei@buaa.edu.cn.

This work is supported in part by the National Natural Science Foundation of China (Grant No. 62072019), the Beijing Natural Science Foundation (Grant No. L243031), and the National Key R&D Program of China (Grant No. 2023YFB4503704 and 2024YFB4505601).

The `Finesse` framework is avaiable at https://github.com/BUAA-CI-LAB/Finesse.

ISCA '25, Tokyo, Japan
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731033

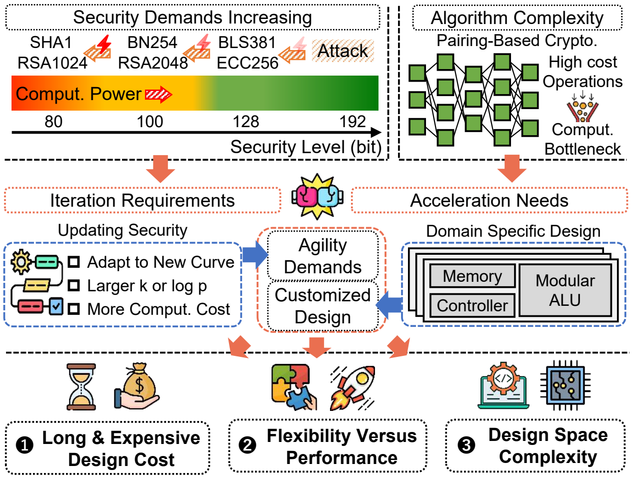


Figure 1: The challenges of PBC accelerator design.

ranging from CPUs [6, 7], GPUs [8], to server-side FPGAs [9] and ASIC [10]. Among these, FPGAs and ASICs excel primarily due to their support for domain-specific designs [11], enabling customized datapaths and logic units that achieve low latency and high resource efficiency in terms of area and power for pairing accelerators.

However, research in this area faces several critical challenges, as illustrated in Figure 1. The security level of pairings is not static; it diminishes as attack methods improve and as computational power advances [12–15]. The growing diversity of application requirements in pairing-based cryptography (PBC) necessitates iterative updates to maintain security and performance standards. This ongoing need for adaptation gives rise to **challenge ❶: the PBC hardware re-engineering costs**.

Existing works with high performance efficiency are designed with specific pairing parameters in mind. While these designs achieve good results, they also lead to high re-engineering costs. On the other hand, works with higher flexibility suffer from low performance due to a lack of parallelism in their architecture, and their flexibility is constrained by the specific hardware implementation (e.g. difficulty supporting curves with larger bit widths), which also results in unavoidable re-engineering costs. In summary, adapting existing works to meet the increasing security demands while maintaining high performance requires considerable repetitive architectural design work. In fact, this constitutes **challenge ❷: the absence of an efficient abstraction system that provides unified support for arbitrary pairing curves.** Introducing such an abstraction can decouple the design process, thus enhancing both the design flexibility and performance potential. However, it also brings **challenge ❸: hierarchical operator variants and architecture co-design complexity**. The interdependence among software algorithms, field operator choices (arithmetic method selections), and hardware configurations creates a complex design space. Optimized operator variants shows different speed comparision results on different hardware architectures. Existing frameworks often overlook this complexity, resulting in lost opportunities for optimization and adaptability[9, 10, 16–18]. Addressing these challenges requires a comprehensive understanding of the design space to enable effective design and improvement of PBC systems.

In this paper, we introduce Finesse, an agile design framework for PBC. Central to its methodology is the utilization of an abstraction system. This abstraction framework bridges high-level algorithms and hardware designs through clear representations and optimized mappings, providing unified support for arbitrary pairing curves in response to increasing security demands.

Through the expressive abstraction system, Finesse's design flow has integrated a parameterized hardware architecture, an advanced compiler and a simulator to form an effective co-design cycle, enabling simple design space exploration within the complex design space.

The results from the Finesse framework are significant. Comprehensive evaluations show substantial performance improvements, achieving 34× throughput and 6.2× slice efficiency compared to previous flexible frameworks, and 3× throughput and 3.2× area efficiency over state-of-the-art ASIC accelerators. Finesse compiles code in minutes, accelerating the development cycle while ensuring that the resulting accelerators are robust and optimized for the evolving demands of modern cryptography.

To summarize, this paper makes the following key contributions:

- We propose the Finesse design framework for pairing-based cryptography, enhancing agility in accelerator design by automating significant portions of the lengthy algorithm-to-hardware flow.
- We developed an abstraction system that includes IR, ISA and hardware models to support the framework.
- To the best of our knowledge, we are the first to incorporate **co-design mechanism** into such a framework, allowing for the effective exploration of the complex relationship within the design space.
- We presented a comprehensive evaluation of Finesse on a prototype implementation, focusing on five key aspects of the framework: accelerator efficiency, framework scalability, compilation, co-design, and agility.

2 Background and Motivation

2.1 Pairing Calculation Panorama

Pairing is a bivariate function satisfying linearity in each of its arguments independently. Optimal Ate pairing [19] over elliptic curve is the de facto standard of pairing construction technique in the cryptography community, surpassing Weil pairing and Tate pairing [20] due to its computational efficiency. The optimal Ate pairing $e(P, Q) : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ for curve E has a complex general form, which is essentially a rational function characterized by curve-determined parameters, constructed with evaluations of the line function ℓ at point P with respect to multiples of point Q. Key parameters and notations are briefly listed in Table 1.

Pairings are defined over pairing-friendly curves to ensure computational practicality. Among these, BN [21] and BLS [3] are the most widely utilized curve families. Examples of pairing-friendly curves are characterized in Table 2.

Pairing Calculation. The optimal Ate pairing algorithm for BN and BLS curves is introduced in Algorithm 1. An overview of group

Table 1: Symbols and notations in pairing calculation.

Notation	Description
p	size of base field
k	embedding degree of curve
r	pairing group order
$\mathbb{F}_p$	base prime field of size p
$\mathbb{F}_{p^k}$	extension field of size p^k, k-th extension of $\mathbb{F}_p$
$E[F]$	curve group, point coordinates in field F
$\mathbb{G}_1$	pairing 1st source group ($E[\mathbb{F}_p]$)
$\mathbb{G}_2$	pairing 2nd source group ($E[\mathbb{F}_{p^k}]$ or $E'[\mathbb{F}_{p^{k/6}}]$)
$\mathbb{G}_T$	pairing target group ($\mathbb{F}_{p^k}$)
$\ell_{Q_1,Q_2}(P)$	(tangent) line function
$e(P,Q)$	pairing function, $\mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$
$\mathtt{M}, \mathtt{S}, \mathtt{A}, \mathtt{B}$	multiplication, squaring, addition, adjunction
$\mathtt{PA}, \mathtt{PD}$	point addition, point doubling

Table 2: Examples of pairing-friendly curves.

Curve	Param. (bit)				Security
	$\log\lvert t\rvert$	$\log p$	$\log r$	$k \log p$	(bit)
BN254	62	254	254	3039	100
BN462	114	462	462	5535	130
BN638	158	638	638	7647	153
BLS12-381	64	381	255	4569	123
BLS12-446	75	446	299	5352	130
BLS12-638	109	638	427	7656	148
BLS24-509	51	509	408	12202	192

Table 3: Costs in pairing calculation.

Group	Storage	Operation Costs
$\mathbb{F}_p$	$(\log p)$ bit	$\mathtt{M}_1, \mathtt{S}_1 \in O(\log^{1.58} p)$ $\quad \mathtt{A}_1, \mathtt{B}_1 \in O(\log p)$
$\mathbb{F}_{p^{2d}}$	$2\,\mathbb{F}_{p^d}$	$\mathtt{M}_{2d} = 4\mathtt{M}_d\ 2\mathtt{A}_d\ 1\mathtt{B}_d$ or $3\mathtt{M}_d\ 5\mathtt{A}_d\ 1\mathtt{B}_d$ or ...
$\mathbb{F}_{p^{3d}}$	$3\,\mathbb{F}_{p^d}$	$\mathtt{M}_{3d} = 9\mathtt{M}_d\ 6\mathtt{A}_d\ 2\mathtt{B}_d$ or $5\mathtt{M}_d\ 33\mathtt{A}_d\ 2\mathtt{B}_d$ or ...
$E[\mathbb{F}_{p^d}]$	$3\,\mathbb{F}_{p^d}$	$\mathtt{PA}_d = 11\mathtt{M}_d\ 5\mathtt{S}_d\ 13\mathtt{A}_d$ or $12\mathtt{M}_d\ 25\mathtt{A}_d$ or ... $\mathtt{PD}_d = 2\mathtt{M}_d\ 5\mathtt{S}_d\ 11\mathtt{A}_d$ or $5\mathtt{M}_d\ 6\mathtt{S}_d\ 11\mathtt{A}_d$ or ...

Algorithm 1: Optimal Ate Pairing (BN/BLS)

```
Input: $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$
Output: $e(P,Q) \in \mathbb{G}_T$
if curve family is BN then
    $u \leftarrow 6t + 2$
else if curve family is BLS then
    $u \leftarrow t$
$(T, f) \leftarrow (Q, 1)$
for $i \leftarrow \lfloor \log u \rfloor$ downto 0 do          /* Miller Loop */
    $(T, f) \leftarrow ([2]T, f^2 \cdot \ell_T(P))$
    if $u[i] = 1$ then
        $(T, f) \leftarrow (T + Q, f \cdot \ell_{T,Q}(P))$
if curve family is BN then
    $Q_1 \leftarrow \mathrm{frob}(Q)$
    $Q_2 \leftarrow -\mathrm{frob}(Q_1)$
    $(T, f) \leftarrow (T + Q_1, f \cdot \ell_{T,Q_1}(P))$
    $(T, f) \leftarrow (T + Q_2, f \cdot \ell_{T,Q_2}(P))$
$f \leftarrow f^{(p^k-1)/r}$                 /* Final Exponentiation */
return $f$
```

computation costs and their optimized alternatives summarized from [22, 23] is presented in Table 3. The algorithm consists of two main components: the Miller loop and final exponentiation, which account for approximately 40% and 60% of the overall computation cost, respectively.

The Miller loop involves iterative calculation of values of the Miller function through point doublings and additions, as well as evaluations of line and tangent functions. Common optimization strategies for the Miller loop include the use of non-adjacent forms and the integration of point operations with line operations.

The final exponentiation transforms Miller function values into canonical form, ensuring consistency by resolving coset equivalence relation. Key optimizations include Frobenius-based techniques for the easy part and decomposition and reuse strategies for the hard part, as demonstrated by [24] and [25]. Additionally, operations within the cyclotomic subfield are optimized to further improve efficiency.

The security of a given pairing diminishes as cryptographic attack algorithms evolve, while computational power continues to increase. To maintain a suitable security level (128/192/... bits), applications are adopting curves with progressively wider bit-widths. Additionally, to preserve the balance between the hardness of the FFDLP/ECDLP while keeping bit-width in a reasonable range, the embedding degree must also increase [20]. Although different curve families share commonalities in terms of extension fields, twist curves, and the fundamental algorithmic framework, their computational details vary significantly.

Insights. Pairing has applications in various cryptographic fields. In the Groth16 [5] zero-knowledge proof system, pairing is used to verify the correctness of a proof by checking whether an equation involving a bilinear relation holds. Pairing significantly reduces the proof size, making verification more efficient.

The key aspects of pairing computation lie in bit-width, extension field arithmetic and reduction costs dominated by embedding degree and twist degree, as well as the optimization methods applied to the final exponentiation. As these requirements evolve, pairing computations tend to introduce wider multiplication widths, more complex control flow, greater pressure on memory access patterns.

2.2 Motivations

❶ ***Mitigate Re-engineering Costs to Keep Pace with Growing Cryptographic Demands.*** Designing accelerators for bilinear pairings has long been a complex and resource-intensive endeavor. The inherently multi-layered nature of pairing computations often makes structural approaches a natural choice, as it encourages designers to map algorithmic hierarchies directly into circuit hierarchies [16, 26]. Alternatively, some works resort to manually

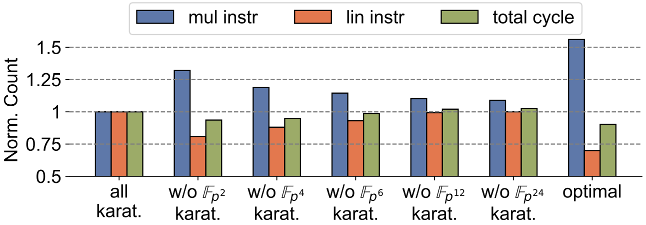


Figure 2: Comparison of operator-level optimization combinations. Curve: BLS24-509, Algo: O-Ate.

transcribing algorithms into microcodes [9], handling each operator at a fine-grained level. While both approaches can achieve functional correctness, they commonly require extensive human intervention in tasks such as operator decomposition, control signal generation and operation scheduling, resulting in substantial design costs and extended development cycles. In summary, expanding the scope of these works to support a broader range of pairing curves or updating them would present significant challenges.

Furthermore, the structural differences across pairing families and curve types highlight the need for greater agility in the design process. Agility enables designers to adapt quickly to diverse requirements, reducing the overhead of manual re-engineering, and accelerating the design cycle. By fostering faster prototyping and iterative feedback, **an agile framework addresses the inefficiencies of traditional methods and empowers pairing accelerator designs to keep pace with evolving cryptographic demands**.

❷ ***Propose an Efficient Abstraction to Bring Performance and Flexibility***. In light of growing needs, recent works have stumbled around the threshold of flexibility. FlexiPair [17] is a flexible pairing framework that aims to provide flexibility for edge devices at the expense of performance potential, revealing significant limitations in its approach. Its framework employs a fixed hardware architecture, which lacks hardware abstraction and co-design capabilities, resulting in limited extensibility. Performance improvements are constrained primarily by bottlenecks in memory and ALU operations. Furthermore, the optimization strategies employed are insufficient, neglecting the potential for software compilation optimizations and software-hardware co-optimization.

On the other side, traditional accelerator designs have completely ruled out the option for flexibility. The current SOTA accelerator on ASIC [10] serves as a prime example of this limitation. Its customized computational layers, particularly the ALU specialized for $\mathbb{F}_{p^2}$, are not adaptable to non-$\mathbb{F}_{p^2}$ curves, thereby limiting its versatility. Additionally, their optimization methodology for customized pipeline structure focuses solely on a narrow scheduling window, mapping $\mathbb{F}_{p^{12}}$ to $\mathbb{F}_{p^2}$, which ignored the potential of global optimization.

To achieve the coexistence of high performance and flexibility, an effective abstraction system is needed to fully cover the design process. Such an abstraction should embrace both algorithmic and architectural possibilities, supporting various curves, operators, and hardware components in a compatible and extensible manner. To summarize, **a compatible and extensible abstraction is the common soil for achieving both flexibility and high-performance in overall architecture**.

❸ ***Exploring the Complex Design Space for Optimization***. We begin by examining a small experiment related to Karatsuba optimization over finite fields. Karatsuba is an optimization technique aimed at reducing the number of multiplications, at the cost of increasing the number of linear operations. While this method proves effective on platforms like CPUs, its benefits may not translate as clearly to hardware accelerators.

This discrepancy arises because accelerators typically access memory with a width that directly matches the base field bitwidth. Both linear and multiplication operations exert the same memory bandwidth pressure, but linear operations perform less computation per memory access. Consequently, linear operations result in lower computational throughput per memory load. Furthermore, on single-issue architectures, when both types of operations occupy one full cycle in the pipeline, the increased number of linear operations exacerbates this imbalance in the instruction issue queue.

However, for higher-degree fields (i.e. $\mathbb{F}_{p^{12}}$ or $\mathbb{F}_{p^{24}}$), the advantage of Karatsuba method cannot be ignored. High-level multiplications are decomposed into more $\mathbb{F}_p$ instructions (k^2 or $k^{1.585}$, while linear operations only break down into k $\mathbb{F}_p$ instructions. As shown in Figure 2, we conducted a validation experiment on a basic single-issue architecture. By disabling Karatsuba-like optimizations in the $\mathbb{F}_{p^2}$ or $\mathbb{F}_{p^4}$ operators, we observed a reduction in the overall cycle count, compared to using optimization on all levels.

The impact of algorithmic optimizations varies significantly depending on the underlying hardware design, demonstrating that effective DSE (design space exploration) is crucial for uncovering configurations that maximize performance while efficiently utilizing hardware resources.

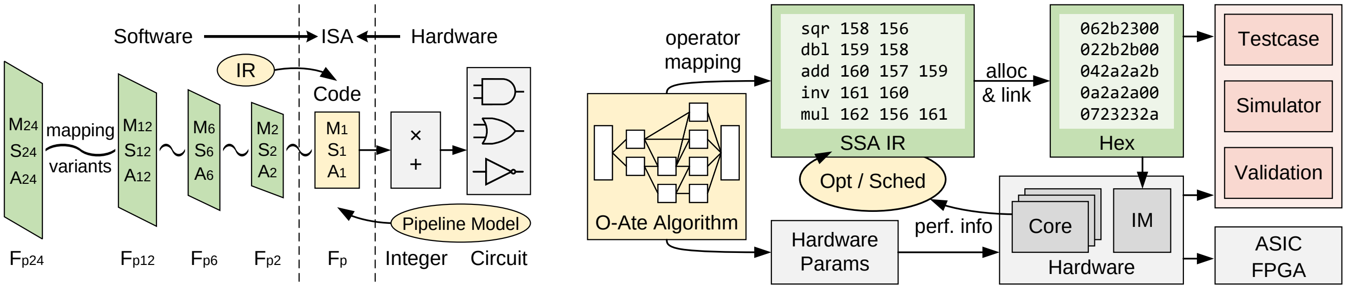
The design space for pairing accelerators is shaped by factors like operator variants, IP availability and hardware data path structures, requiring careful coordination between components and architecture. Diverse and often conflicting goals—such as optimizing area, throughput, or area/delay trade-offs—add complexity to the exploration. In this context, **DSE and co-design frameworks are crucial, enabling systematic evaluation of design options and hardware-software co-optimization to unleash the potentials of pairing accelerators**.

3 Finesse Framework

3.1 Framework Overview

As sketched in Figure 3, `Finesse` is an agile full-process design framework for pairing-based cryptography, offering a comprehensive and extensible system that spans from high-level algorithm description to low-level hardware models, providing direct validation and deployment across various cryptographic curves. `Finesse` comes with a fully functional implementation, providing out-of-the-box support for popular curves.

Methodology. Pairing accelerator design is inherently a multi-layered problem, spanning from high-level finite field operators (e.g. $\mathtt{M}_{24}$) and point operators (e.g. $\mathtt{PA}_4$) down to low-level circuit design.



```
sqr 158 156
dbl 159 158
add 160 157 159
inv 161 160
mul 162 156 161
```

```
062b2300
022b2b00
042a2a2b
0a2a2a00
0723232a
```

(a) Methodology **(b) Design flow**

Figure 3: Overview of Finesse design framework.

This hierarchical structure introduces significant challenges in mapping algorithms efficiently to hardware. To address this, Finesse adopts a co-design methodology that strategically partitions software and hardware layers while providing abstractions tailored to each level of the computation.

The boundary between software and hardware is defined by an instruction set architecture (ISA) abstraction. This abstraction standardizes interactions between algorithmic logic and hardware, serving as a modular interface. Above the ISA, high-level computations are represented through an intermediate representation (IR). The IR enables software optimizations, such as instruction scheduling and dependency analysis, while maintaining flexibility for different hardware backends. Below the ISA, hardware implementations rely on pipeline models to describe its behavior, including latency, memory and instruction issue parameters. These models guide the hardware realization process, ensuring alignment with upstream abstractions while permitting design variations. By integrating these abstractions, Finesse establishes a systematic framework for software/hardware co-design, supporting pairing computations across diverse configurations.

Design Flow. The design flow of Finesse is organized into a set of modular components, each addressing specific aspects of pairing accelerator development. The hardware and simulator part uses abstraction to define and implement pipeline behavior, enabling rapid exploration of configurations. The compiler and optimization part maps high-level operators into IR, supporting co-design between software and hardware. It applies scheduling and data flow optimizations while enabling basic DSE, which systematically evaluates performance and resource trade-offs. The validation part verifies correctness and performance on simulators and prototype platforms (ASIC and FPGA), generating feedback for iterative design refinement. Finesse also offers a basic operator kit for quickly porting new curves or pairing algorithms into the framework. These components work together to streamline the development process and support a wide range of pairing accelerator configurations.

3.2 Abstraction

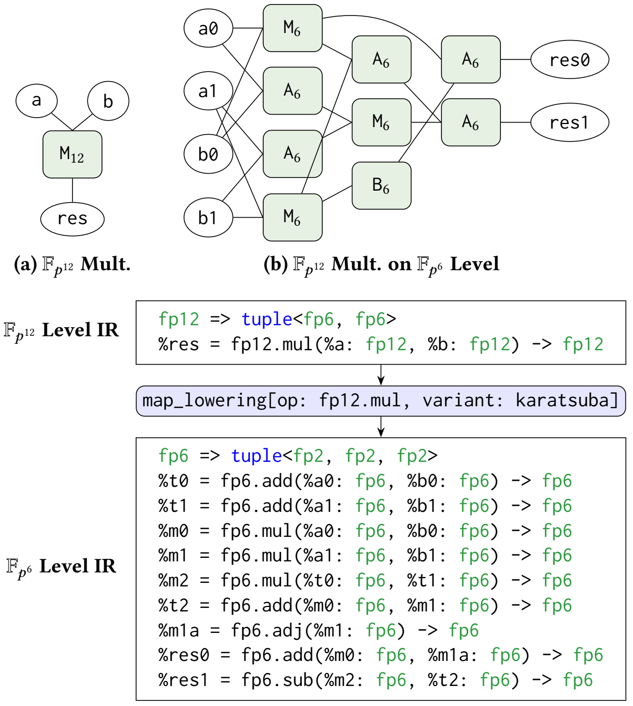
Abstraction in the Finesse framework defines clear interfaces between software and hardware layers, enabling efficient mapping of high-level algorithms to hardware. It utilizes intermediate representations (IR), instruction set architectures (ISA), and hardware models to optimize and flexibly represent both software and hardware components. This structured approach ensures interoperability, scalability, and streamlined co-design, supporting diverse pairing accelerator configurations.

Table 4: IR operations and their supported argument data type. `fp-like` = `fp` or `fpd`, `ep-like` = `ep` or `epd`. Adjoined element refers to the element adjoined to the base field for defining the extension field.

IR Op.	Description	Supp. Arg. Type
`add/sub`	field addition/subtraction	`(fp-like, fp-like)`
`muli`	field scalar multiplication	`(int, fp-like)`
`mul`	field multiplication	`(fp-like, fp-like)`
`sqr`	field squaring	`fp-like`
`exp`	field exponentiation	`(fp-like, int)`
`adj`	multiply by adjoined el.	`fpd`
`conj`	conjugate to adjoined el.	`fpd`
`frob`	Frobenius endomorphism	`(fp-like, int)`
`padd`	curve point addition	`ep-like`
`pmul`	curve scalar multiplication	`(int, ep-like)`

Challenges. Main consideration in abstraction design is focused on compatibility and extensibility. On the algorithm side, abstraction needs to support the family/curve/operator triplet by capturing commonalities and covering diverse PBC primitives. We opted to keep it as simple as possible to preserve extensibility towards broader fields in cryptography. On the hardware side, Finesse needs to be compatible with a series of architectural candidates, a wide portfolio of ALUs and memory units ranging from open-source free designs, self-made designs and available proprietary IP cores. Also Finesse should be able to evolve and support novel datapaths that provides better parallelism in the future. This can be achieved through ISA-level extensions and improvements on instruction selection strategies. Facing these challenges, Finesse has chosen to define abstractions carefully and move complexities to above ISA level rather than sub-ISA level, solving the problem mainly at compile time.

Abstraction Design. Finesse IR is mainly focused on expressing calculation on algebraic objects. Custom data types include: `fp`



(a) $\mathbb{F}_{p^{12}}$ Mult.

(b) $\mathbb{F}_{p^{12}}$ Mult. on $\mathbb{F}_{p^6}$ Level

$\mathbb{F}_{p^{12}}$ Level IR

```
fp12 => tuple<fp6, fp6>
%res = fp12.mul(%a: fp12, %b: fp12) -> fp12
```

```
map_lowering[op: fp12.mul, variant: karatsuba]
```

$\mathbb{F}_{p^6}$ Level IR

```
fp6 => tuple<fp2, fp2, fp2>
%t0 = fp6.add(%a0: fp6, %b0: fp6) -> fp6
%t1 = fp6.add(%a1: fp6, %b1: fp6) -> fp6
%m0 = fp6.mul(%a0: fp6, %b0: fp6) -> fp6
%m1 = fp6.mul(%a1: fp6, %b1: fp6) -> fp6
%m2 = fp6.mul(%t0: fp6, %t1: fp6) -> fp6
%t2 = fp6.add(%m0: fp6, %m1: fp6) -> fp6
%m1a = fp6.adj(%m1: fp6) -> fp6
%res0 = fp6.add(%m0: fp6, %m1a: fp6) -> fp6
%res1 = fp6.sub(%m2: fp6, %t2: fp6) -> fp6
```

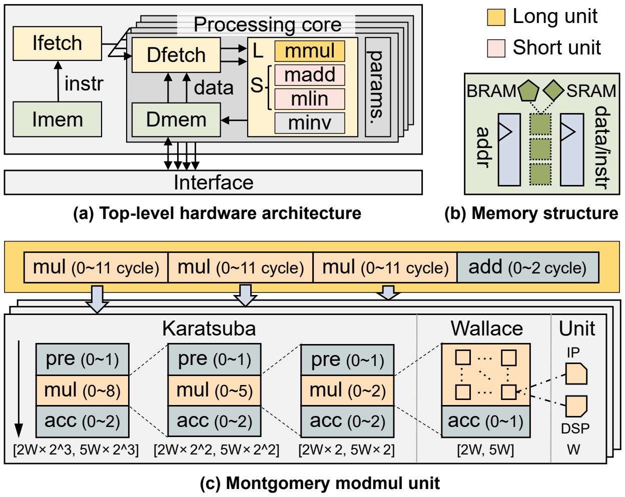
(c) Cross-layer IR mapping with variant specification

Figure 4: Example of Finesse's operator mapping through IR abstraction.

→ $\mathbb{F}_p$, fpd → $\mathbb{F}_{p^d}$, ep → $E[\mathbb{F}_p]$, epd → $E[\mathbb{F}_{p^d}]$, where d denotes the dimension of the extension field relative to base field $\mathbb{F}_p$. Necessary parameters (refer to Table 1 for a list) determining the field structures and curve structures is incorporated as attributes to the IR. Table 4 gives a list of defined operations on these objects. As a simplification, operations between fp-like objects or ep-like objects requires divisibility on their dimension parameters d (or else an efficient homomorphism would be required, which is possible, but over-complicates the abstraction system).

Figure 4 illustrates an example of Finesse's abstraction system, showcasing the transformation of a high-level $\mathbb{F}_{p^{12}}$ multiplication operation into a lower-level $\mathbb{F}_{p^6}$ representation through IR. The upper part of the figure depicts the $\mathbb{F}_{p^6}$ operation expressed using abstract operators, while the lower part demonstrates the detailed decomposition into $\mathbb{F}_{p^6}$ operators, using the Karatsuba variant. Following this approach the framework bridges high-level algorithmic constructs with hardware-aware granular operations, ensuring efficient and modular design.

Finesse defines a simple RISC-flavor $\mathbb{F}_p$-level ISA with VLIW extension. Machine operations include: linear operations (NEG, DBL, TPL, ADD, SUB), multiplicative operations (SQR, MUL), inverse operation (INV), miscellaneous (NOP, CVT, ICV). Finesse performs computation in its dedicated on-chip/on-fabric register banks, thus all operands are registers. CVT and ICV operations are designed for post/pre I/O data format conversions. In VLIW extension, multiple operations can be packed into a single "wide instruction", or "issue slot", to enable explicit ILP.



(a) Top-level hardware architecture

(b) Memory structure

(c) Montgomery modmul unit

Figure 5: Hardware architecture of Finesse framework.

Finesse's hardware model describes hardware resource information and instruction itineraries. These include: number of linear ALUs, number of register banks and register quota per bank, simultaneous read/write capacity for each bank, presence of write back ring buffers for register banks, maximum operations allowed in a single VLIW instruction (similar to that of issue width in multi-issue architecture), delay info and resource consumption for each operation (itineraries). Currently Finesse asserts a few reasonable constraints on the model: at most 1 mmul ALU per core, at least many register banks as VLIW width, at least 2 reads + 1 writes per bank per cycle, existence of write back ring buffers on VLIW architectures (width ≥ 2).

Abstraction Overhead. In Finesse, abstraction above ISA layer does not involve complex programming paradigm, and is essentially zero-cost, i.e. no additional control information stored at runtime. When decomposed onto $\mathbb{F}_p$ level operations, higher-level constuct info can be dropped, and constants needed in lowering mapping can fit in a small table to be fetched at runtime. Assertions in and below ISA inevitably hide some possible optimization opportunity, but this tradeoff is worthy in a systematic approach for agile co-design.

3.3 Hardware Architecture

With abstraction decoupling the software and hardware, the hardware only needs to focus on implementing the pipeline model architecture, which is designed to support operations over $\mathbb{F}_p$. Figure 5 provides an overview of the basic hardware architecture supported by our Finesse framework. As shown in Figure 5(a), the architecture is built on a pipeline structure, consisting of instruction memory/fetch units, and one or more processing cores, each of which includes data memory/fetch units and an ALU. The ALU features four modular arithmetic units designed for $\mathbb{F}_p$ operations, including modular multiplication, modular addition, modular doubling (in the mlin unit), and modular inversion, among others.

We employed the Jacobian coordinate system to implement the pairing algorithm, which requires modular inversion (performed in the minv unit) only once. Consequently, the relatively complex minv unit is designed using an iterative structure, while all other computation units adopt a fully pipelined structure. In our prototype, modular multiplication is treated as a `Long` pipeline unit, whereas other linear units are considered `Short` pipeline units.

Parameterization. Through parameterized design, the architecture can be adapted to diverse application requirements and platform constraints, while providing a foundation for performance optimization. The primary parameters include curve constants, data width, memory configuration, the number of parallel cores, base unit mapping to various IP portfolios, and the pipeline depth of computation units. Key benefits of parameterization include:

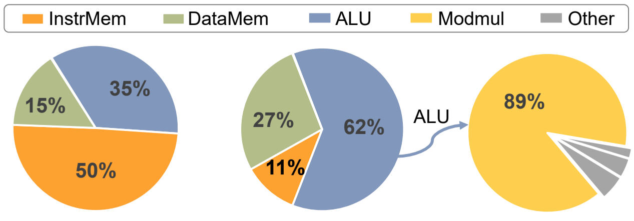
- **Adaptability** to different curve scenarios via parameterization of curve constants, data width, and memory size.
- **Flexibility** in adapting to varying throughput requirements through adjustment of the parallel core count.
- **Platform independence** enabled by the mapping of base units to different deployment platforms (e.g. ASIC, FPGA).
- ALU family co-design optimization is supported by parametrizing the **pipeline depth** of key computational units.

Multi-core. As shown in Figure 6, instruction memory occupies 50% of the area in a single-core design. Analysis reveals that, for pairing computations on the same curve, the operations are identical. This consistency allows us to replicate multiple data memory and ALU while utilizing a shared instruction memory. Figure 6(b) presents the area breakdown of an 8-core design, where instruction memory accounts for only 11% of the total area. This reduction highlights better area utilization, as the total area increases by 4.5 times while achieving an 8-fold improvement in overall throughput, resulting in a 77% gain in area efficiency. In fact, in our architectural design, the number of parallel cores and area efficiency aligns with Amdahl's law.

Indeed, this parallel approach aligns with the SIMT architecture, which meets the demands of high-throughput applications and further enhances area efficiency.

Optimizations. It is important to note that the area and timing are directly influenced by parameterization. Firstly, the design must ensure that the storage and computation units support different data width, and the computation units support pipeline depth parameterization. Secondly, from a co-design perspective, the pipeline depth of the ALU itself affects the overall timing performance (refer to Figure 11 in the below). To enhance the efficiency of pairing operations, timing and area optimizations were applied to parameterized storage and computation units.

For storage units, the design enables arbitrary bitwidth and depth by automatically combining small basic memory units into larger configurations. As depicted in Figure 5(b), to reduce path delay caused by combining smaller memory units into larger ones, registers are placed before and after the memory, creating a three-stage pipeline for read/write operations. The attributes of the basic memory block are fixed by IP vendors. So the storage area (i.e., the number of basic memory block used) is more dependent on the binary size the compiler generates (for IMem) and the maximum number of active registers (for DMem). Meanwhile, the basic memory units can be mapped to specific platform primitives.



(a) 1-core, 1.77 mm^2

(b) 8-core, 8.00 mm^2

Figure 6: Hardware area breakdown. Hardware model: `Long` = 38cy, `Short` = 8cy, `2R1W`/cy. Curve: BN254N, Algo: O-Ate.

As illustrated in Figure 6(b), Modular multiplication (mmul) unit is the core of the ALU, which accounts for 89% of the ALU's area. Our primary objective is to optimize mmul while achieving parameterization of bitwidth and depth. In terms of timing, we adopt a deeply pipelined design, decomposing modular multiplication into multiple stages to achieve high throughput. Regarding area, we reduce the number of multiplications by leveraging the Karatsuba algorithm. To facilitate optimization within a parameterized framework, we design a hierarchical modular multiplication module, as illustrated in Figure 5(c). At the basic unit layer, the multiplication width W determines the critical delay path, as it is directly mapped to FPGA DSP blocks or ASIC multiplier IPs. To enable efficient module partitioning, we encapsulated $2W$ to $5W$ bit multiplier modules using the Wallace tree algorithm based on the basic unit. Further optimized by recursively applying the integer Karatsuba multiplication algorithm n times, the structure covers a range from $2W \times 2^n$ to $5W \times 2^n$, which effectively reduces the multiplier's area. For instance, with $W = 16$ and $n = 3$, the proposed approach achieves an approximate 40% reduction in area compared to naive multiplication.

3.4 Simulator

The simulator empowers both software and hardware validation flows. On the algorithmic side, to enable validation of post-compile code execution trace, we have implemented a single-cycle functional simulator capable of executing SSA instructions. The correctness verification is accomplished through cross-validation of computational results against established cryptographic libraries such as MCL [27], MIRACL [28], and RELIC [6].

At the hardware level, we have implemented a cycle-accurate simulator consistent with the RTL behavior based on the pipeline model to simulate instruction delay and data dependence. Within our framework, this simulation platform serves as a experimental infrastructure, providing data references for works like compiler affinity optimization and design space exploration.

3.5 Compilation Techniques

Compilation Pipeline. In contrast to modern general-purpose compilers, `Finesse` uses a shorter compilation pipeline, following the execution order:

- **CodeGen**: Simplified leveraging algorithmic characteristics. In the optimal Ate pairing algorithm, both the Miller loop and final exponentiation have fixed loop parameters based on the underlying curve, allowing convenient loop unrolling and

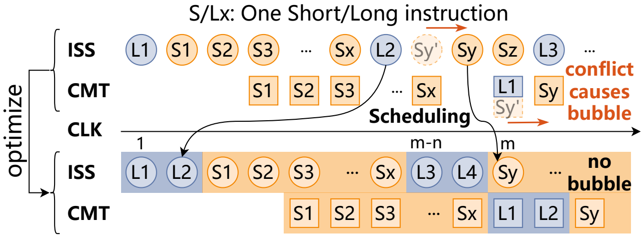


Figure 7: Illustrated example of instruction issue slot affinity optimization. `Long` = $m$cy, `Short` = $n$cy. `L1` and `Sy'` are $(m-n)$cy apart, causing latter to stall until `Sy`. Setting issue slot affinity helps compiler to avoid this issue.

restructuring into a single basic block. The code generation task is completed by mapping the algorithm into IR, with respect to operator variants.

- **IROpt**: Standard SSA-based data flow optimizations, with additional assumptions from finite field arithmetic. Specifically, constant propagation with Frobenius constant tables, strength reduction, dead code elimination and global value numbering using commutativity on finite fields.
- **BankAlloc**: Assigns values to register banks. A simple residual assignment strategy serves as an effective baseline.
- **PackSched**: Schedules and packs operations into issue slots. Supports both single issue architectures and VLIW architectures.
- **RegAlloc**: Sequential register allocation within banks, based on liveness analysis.
- **ASM**: Translates IR into hardware-defined instruction encoding.
- **Link**: Consolidates various basic blocks into a single binary and resolves entry address offsets.

Scheduling Strategies. `Finesse` supports scheduling strategies for both single-issue and VLIW architectures. Scheduling is performed in a single pass immediately after BankAlloc, where values are assigned to register banks without being mapped to specific registers yet. `Finesse` uses a top-down scheduling algorithm with affinity-based selection order and constraint solving for register bank read/write constraints.

The core innovation of `Finesse` scheduling strategy lies in its *issue slot affinity optimization*. Figure 7 provides a conceptual example of this approach. From a hardware abstraction perspective, R/W operations in memory blocks are constrained by certain limitations, and conflicts can arise when a `Long` instruction is issued followed by a `Short` instruction after a certain delay. To address this, `Finesse` partitions the issue slots into periodic intervals based on the difference in cycle counts between `Long` and `Short` instructions. Within each interval, specific positions are assigned `Long` instruction affinity, while the rest are assigned `Short` instruction affinity. Following notation of Figure 7, we can formulate the affinity for issue slot at cycle T as

$$\text{Affinity}(T) \coloneqq \frac{T \bmod (m-n)}{m-n} \leq \frac{\#\texttt{LongInstr}}{\#\texttt{Instr}} + \beta,$$

where β is a tunable parameter, and $\text{Affinity}(T) = \texttt{True}$ implies `Long` affinity, `Short` affinity otherwise.

Algorithm 2: Operation Packing and Scheduling

```
Input: IR code $\mathcal{P}$, in SSA form
Output: A valid schedule $\mathcal{S}$, satisfying HW constraints
$\mathcal{S} \leftarrow [\,]$
$deps \leftarrow \{\,\}$
$trigger \leftarrow \{\,\}$
$queue \leftarrow [\,]$
Function solveMaxValidInstrPack($now$)
    // state: (maxInst, candInst)
    $f \leftarrow \{\varnothing : (0, [\,])\}$
    $bestState \leftarrow \{\,\}$
    $issueQueue \leftarrow$ collect readied instr from $queue$ at $now$
    for $instr$ in sortByAffinity($issueQueue$) do
        $state_1 \leftarrow$ compress itinerary of $instr$ to DP state
        for $state_2 \in f$ do
            if $state_1$ does not contradicts $state_2$ then
                $new \leftarrow state_1 \cup state_2$
                update $f[new]$ if $f[state_2] + instr$ is better
                update $bestState$ by $f[new]$
    return $f[bestState]$.candInst
for $instr$ in $\mathcal{P}$ do
    $deps[instr] \leftarrow$ countDependency($instr$)
    if isConstOp($instr$) then
        $\mathcal{S}$.append($instr$)
    else
        $trigger[instr.operand]$.append($instr$)
$issueTime \leftarrow 0$
$n \leftarrow$ len($\mathcal{P}$) − #ConstOp
while $n > 0$ do
    $instrs \leftarrow$ solveMaxValidInstrPack($issueTime$)
    $\mathcal{S} \leftarrow \mathcal{S} + instrs$
    for $instr$ in $instrs$ do
        for $inst$ in $trigger[instr]$ do
            $deps[inst] \leftarrow deps[inst] - 1$
            if $deps[inst] = 0$ then
                $queue \leftarrow queue + [instr]$
        $queue \leftarrow$ delete($queue, instr$)
    $n \leftarrow n -$ len($instrs$)
    $issueTime \leftarrow issueTime + 1$
return $\mathcal{S}$
```

`Finesse` employs a constraint solving process combining data dependency, issue slot affinity and instruction itineraries to produce a correct and efficient schedule for the IR code. Algorithm 2 gives a detailed formulation of this algorithm. It starts by scanning top-down on DAG structure, and on each cycle T, a table of candidate instructions are drawn from those that are ready to be issued at current cycle, following the order determined by Affinity(T). We use dynamic programming to check for maximum combination of operations into a single issue slot without violation of R/W constraints.

Table 5: Examples of operator variants for key extension fields in curve BLS24-509.

Group	Op.	Variants
$\mathbb{F}_{p^6}$	$\mathtt{M}_6$	Karatsuba, Schoolbook
	$\mathtt{S}_6$	CH-SQR{1,2,3} [29], Complex, Schoolbook
$\mathbb{F}_{p^{12}}$	$\mathtt{M}_{12}$	Karatsuba, Schoolbook
	$\mathtt{S}_{12}$	Complex, Schoolbook
$\mathbb{G}_2$	$\mathtt{PA}_4, \mathtt{PD}_4$	Jacobian, Projective

3.6 Design Space Exploration

As a first step towards comprehensive DSE, Finesse concentrates on solving the problem in a pivotal subspace of the general design space. The design space in our framework is characterized by two key elements: *operator variants combination* and *hardware model*, both of which significantly influence the performance and resource efficiency of the accelerator.

Operator variants combination (examples in Table 5) defines the mapping rule of higher-level operators into lower-level operators. As analyzed in Section 2.2, its influence should be considered jointly with the hardware model. In Finesse, the ALU configuration determines the computational capacity for executing instructions in parallel. Additionally, instruction issue width and instruction scheduling are critical components shaping the architecture's throughput and latency characteristics.

Our framework accommodates a variety of design directives, offering flexibility in the optimization process. The framework can adjust accordingly, whether the focus is on a single objective, such as maximizing throughput or minimizing area, or balancing multiple objectives in a trade-off. This capability ensures that the design process remains versatile, allowing users to define and prioritize their own performance metrics without being restricted to predefined directives. As a result, the exploration process can target performance, efficiency, device resources, or a combination of factors.

The exploration process in our framework is driven by a co-design feedback loop that iteratively refines the hardware-software configuration. This process gathers cycle info from the simulator and hardware metrics provided by EDA tools, enabling continuous optimization of both architectural and algorithmic decisions. Finesse incorporates basic exploration strategies, using exhaustive search for operator variants combinations. Finesse sets the foundation for efficiently navigating the design space, adapting to specific platform-specific constraints and optimization objectives.

4 Evaluation

Framework Implementation. We implemented the Finesse framework in synergy of multiple ecosystems. The compiler and simulator are written in Python, supporting flexible configuration through YAML configuration files and modular invocation with command-line parameters. The parameterized hardware is implemented by SystemVerilog, with its settings automatically read from headers generated by the compilation stack. We also developed a basic operator kit containing elliptic curve operators in both Jacobian and projective coordinates, together with finite field operators from $\mathbb{F}_p$ to $\mathbb{F}_{p^{24}}$ along the finite division lattice of 24. On top of that we implemented 7 curves in 3 curve families as listed in Table 2, and validated correct functionality for all of the resulting accelerators.

Hardware Validation Setup. The experiments were conducted on both ASIC and FPGA platforms.

- **ASIC**: 40nm LP process, using a 1.1 V and a 25 °C typical-typical (TT) library.
- **FPGA**: Xilinx Virtex-7 FPGA, with 108,300 slices, 3,600 DSP blocks, and 1,470 BRAM blocks.

Performance Measurement & Scaling. We use standard EDA toolchain to obtain clock cycles and latency, from which throughput is calculated. The results are precise and time-deterministic, demonstrating that our design is resistant to timing attacks. Experimental results for the FPGA platform are obtained from Vivado tools, including performance and utilization. For the ASIC implementation, performance and area metrics are derived from synthesis using commercial EDA tools. Since different ASIC technology nodes significantly impact performance, area, etc., we refer to [30] and apply equivalent scaling adjustments for these metrics between different technology nodes for ensuring a fair comparison of pairing metrics across various ASIC implementations.

Key Aspects in Evaluation. We evaluate the methods and contributions of Finesse as a framework for designing pairing-based cryptography accelerator by addressing five key aspects:

(1) *Design efficiency*: how efficient is the Finesse design framework?
(2) *Scalability*: is Finesse scalable as security level rises?
(3) *Compilation optimization*: how does Finesse's compile strategies improve pipeline efficiency?
(4) *Co-design*: to what extent can Finesse's co-design mechanism tackle the complexity of design space?
(5) *Agility*: how agile and practical is the Finesse design framework?

4.1 How efficient is the Finesse design framework?

To assess the efficiency of Finesse, we compare our approach with two representative works: [17] and [10], with the details presented in Table 6. The work in [17] utilizes the BN256 curve, while [10] employs the BN254 curve, with both offering an equivalent security level. For consistency, we select the BN254 curve as our test case.

Flexipair [17] stands out for its lightweight, programmable nature, providing high flexibility. However, its limited abstraction constrains extensibility. The lack of abstraction leads to an inability to achieve effective performance exploration. By contrast, Finesse demonstrates a 34× performance improvement, utilizing 5.6× the resources relative to [17] and significantly improving area (slice) efficiency to 6.2×. On the ASIC platform, [10] delivers SOTA performance due to a highly customized ALU design, but sacrifices flexibility. By contrast, Finesse achieves a 3× throughput increase and 3.2× area efficiency improvement in an 8-core configuration.

This comparison shows that the Finesse Framework excels in both performance and area/resource efficiency. Specifically, it not only delivers significant performance gains but also demonstrates

Table 6: Evaluation comparison on the BN254/BN256 curves. Hardware model: `Long` = 38cy, `Short` = 8cy, `2R1W`/cy. Algo: O-Ate.

Work	Platform	Frequency	#Cycle	Latency	Util./Area	Throughput	Throughput/Area
[17]	FPGA Virtex-7	188.5 MHz	2552k	14.14 ms	2506 Slices	70.7 ops	0.028 ops/Slice
Ours (1-core)	FPGA Virtex-7	153.8 MHz	63607	0.413 ms	13928 Slices	2421 ops	0.174 ops/Slice
[10]	ASIC 65nm FDSOI	250 MHz	8487	56.2 µs@1.1 V	12.8 mm^2	17.8 kops	1.39 $kops/mm^2$
Ours (1-core)	ASIC 40nm LP	769 MHz	63607	82.7 µs@1.1 V	1.77 mm^2	12.1 kops	6.83 $kops/mm^2$
Ours (8-core)	ASIC 40nm LP	769 MHz	63607	82.7 µs@1.1 V	8.00 mm^2	96.7 kops	12.09 $kops/mm^2$
Ours (8-core)[1]	ASIC 65nm (equiv.)	423 MHz	63607	150.2 µs@1.1 V	12.0 mm^2	53.3 kops	4.44 $kops/mm^2$

[1] Row has been normalized to be equivalent to 65nm technology from 40nm LP technology [30].

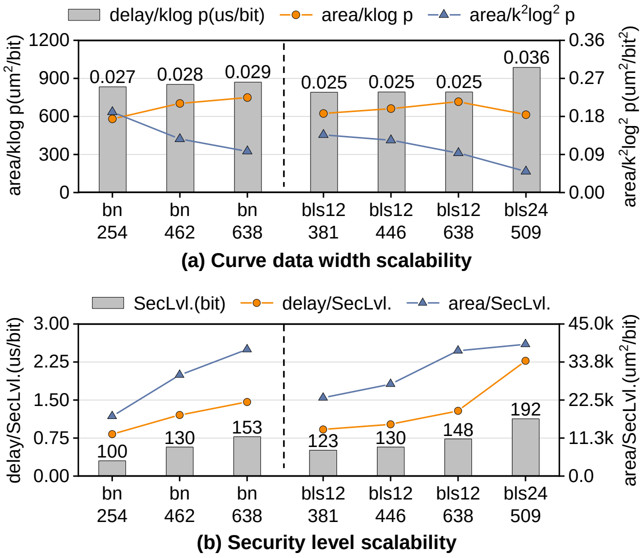


Figure 8: Scalability evaluation of `Finesse` framework. Respectively, $\log p$**, SecLvl. and** k **refer to the base field bit-width, security level and embedding degree.**

Table 7: Evaluation of `Finesse`'s compilation strategies. Compile time ranges from 8.0s/BN254N to 53.1s/BLS24-509. Hardware model: `Long` = 38cy, `Short` = 8cy, `2R1W`/cy. Algo: O-Ate.

Curve	Instr. Reduction Init. → Opt.	IPC Improvement Init. → Opt. (HW 1/2)[1]
BN254N	62.7k → 55.3k (-11.7%)	0.19 → 0.87 / 0.92
BN462	115k → 101k (-12.0%)	0.20 → 0.88 / 0.92
BN638	155k → 137k (-12.0%)	0.20 → 0.88 / 0.92
BLS12-381	81.1k → 74.1k (-8.73%)	0.19 → 0.87 / 0.92
BLS12-446	94.8k → 86.5k (-8.73%)	0.19 → 0.87 / 0.92
BLS12-638	125k → 114k (-8.47%)	0.19 → 0.87 / 0.92
BLS24-509	324k → 271k (-16.4%)	0.22 → 0.88 / 0.97

[1] HW 1/2 refers to the hardware model without/with FIFO buffer, which is an architectural feature alleviating write-back conflicts.

superior resource utilization. In conclusion, the `Finesse` design framework is highly efficient for pairing accelerator design, offering a high performance and resource-efficient solution.

4.2 Is Finesse scalable as security level rises?

Recent progress in number field sieve (NFS) has shaken the security of pairing-based cryptography [31]. To maintain adequate security, cryptography systems must adopt larger bit-width curves and increase the embedding degree, which strains hardware resources.

`Finesse` design framework offers scalability to address these challenges posed by increasing the curve security level. Figure 8 illustrate the performance and scalability of the framework across different curve configurations.

In Figure 8(a), we present the relationship between area, delay, and $k \log p$ for two curve families. The pairing delay exhibits approximately linear growth as the $k \log p$ increases. The ratios of area to $k \log p$ and $k^2 \log^2 p$ are plotted, indicating that despite the increase in computational complexity, the framework controls the area growth to slightly above linear, significantly below the quadratic growth rate anticipated by the complexity of finite field multiplication.

In Figure 8(b), we use the method proposed by Barbulescu and Duquesne [32] to evaluate curve security under the SexTNFS [33] attack. The evaluation results are presented in the gray bar plot, indicating an increase in security level as the $k \log p$ expands. Meanwhile, the line chart indicates that as the security level increases, the ratio of pairing delay to security level remains relatively stable, while the area growth is kept within a reasonable range.

The results suggest that the `Finesse` framework is scalable as security levels increase and can effectively maintain a balance between performance and resource consumption.

4.3 How does Finesse's compile strategies improve pipeline efficiency?

`Finesse`'s compile strategy directly addresses pipeline efficiency through a combination of data flow and architecture-specific scheduling optimizations.

Finding a suitable compilation baseline for **emerging workloads** on a novel customized target accelerator is a non-trivial task. The architectural diversity among accelerators presents a significant challenge for establishing a common compilation baseline. For example, [17] is CISC-like with sub-$\mathbb{F}_p$ ALUs, whereas [10] is FSM-like with $\mathbb{F}_{p^2}$ ALUs. There exists no widely accepted methodology for precisely and fairly comparing instruction count or cycle count across different accelerator architectures. Macro-level comparison of compilation effect is not possible without a common target or established equivalance relation between targets.

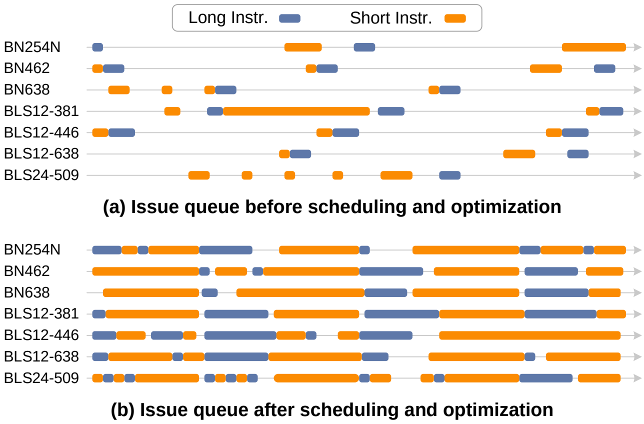


Figure 9: Scheduling and issue slot affinity optimization on issue queue. Each instruction occupies a single slot in issue queue. The snapshot is taken starting from the 10,000th cycle during the simulated execution of SSA IR. Hardware model: `Long` = 38cy, `Short` = 8cy, `2R1W`/cy. Curve: BN254N, Algo: O-Ate.

Our baseline (referred to as "Init." in Table 7, "before" in Figure 9, "Manual" in Figure 10) pairing implementation is built directly from cryptographic literature, i.e. exactly as reported, without alterations that might introduce a favorable bias towards our compiler.

Through data flow optimizations, `Finesse` automates transformations that were previously handled manually in research, such as $\mathbb{F}_{p^k}$ *dense* × *sparse* multiplication. Table 7 quantifies their impact in terms of instruction reduction across multiple curves. Rather than claiming credit for the optimization itself, `Finesse` contributes to agility by performs these optimizations transparently, freeing users from manually handling sparsity.

In terms of scheduling optimization, `Finesse` employs novel scheduling strategies, which include standard code motion based on instruction latencies, together with instruction issue slot affinity optimization. Figure 9 gives a clear visualization in the form of a waterfall chart, illustrating the improved pipeline utilization, showing how pipeline bubbles are minimized. Table 7 contains detailed IPC statistics.

These results have demonstrated that `Finesse`'s optimization strategies significantly improve pipeline efficiency, resulting in notable performance gains in pairing accelerators through reducing execution time and enhancing resource utilization. Collectively, these optimizations provide a solid building block for `Finesse`'s co-design mechanism.

4.4 To what extent can Finesse's co-design mechanism tackle the complexity of design space?

`Finesse`'s co-design mechanism is an initial step towards full-fledged design space exploration. Our work serves as a stepping stone from 0 to 1, paving the way for a more comprehensive approach to optimizing performance across a vast design space.

`Finesse` provides a fully functional implementation capable of performing exhaustive design space exploration. Figure 10 presents results from `Finesse`'s analysis of operator variant combinations and representative pipeline configurations. Unlike the conventional approach of applying Karatsuba optimization at all levels, we explored a variety of operator variants for the BLS24-509 curve, combined with hardware models ranging from a basic single-issue pipeline to a multi-issue pipeline featuring up to 6 linear modular arithmetic units. In addition to typical baseline approaches, we included a manually selected variant combination, guided by heuristics optimized for a single-issue pipeline. With limited parallelism in linear operations this manually tweaked version outperforms typical approach and is near optimal, but with more linear units, all-Karatsuba is still a viable choice.

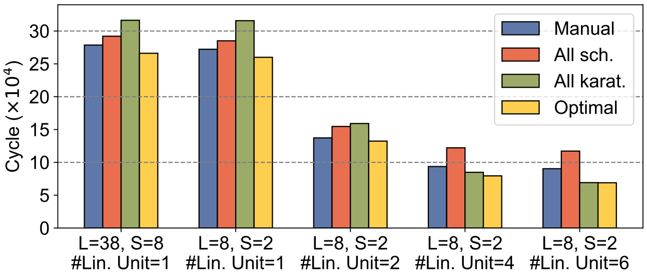


Figure 10: `Finesse`'s design space search on operator variant combinations and representative pipeline configurations. "Manual" refers to combinations of variants we selected manually; "All sch." and "All karat." refer to the combinations entirely using the Schoolbook and Karatsuba variants, respectively; "Optimal" refers to the best combinations obtained from exploration. "L" and "S" denote the cycles for `long` and `short` instructions, and "#Lin. Unit" indicates the number of linear units, which also equals the number of register banks. Hardware model: `Long` = Lcy, `Short` = Scy, `2R1W`/bank/cy. Curve: BLS24-509, Algo: O-Ate.

`Finesse` can also perform co-design with feedbacks from EDA toolchain. Figure 11 reflects the impact of choices regarding the ALU family (ALU family refers to the fully pipelined mmul units with different pipeline depths, which are equivalent to the `Long` instruction cycles described in the paper), with ALU design variations being the primary parameter for optimization. The ALU critical path information is gathered from synthesis results of ASIC toolchain over target technology node, while compile-time estimates of IPC and throughput metrics are obtained from the simulator. These metrics are derived from `Finesse`'s optimization pass, which utilizes hardware abstractions passed in by the co-design mechanism as essential directives in IR scheduling.

The results in Figure 11 indicate a drop in IPC with deeper ALU pipelines, due to the limited inherent parallelizability of the O-Ate pairing algorithm. Additionally, as determined by the constraints of the target technology node, critical paths cease to decrease with deeper pipelines. `Finesse` effectively analyzes this real-world nonlinear relationship, identifying the optimal pipeline depth of 38 cycles on single issue architectures for our experiment setup.

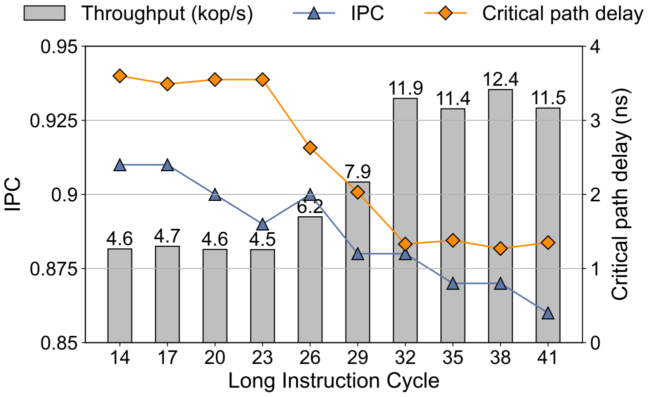


Figure 11: Finesse's co-design mechanism with respect to choices in ALU family. Hardware model: Long = $x$cy, Short = 8cy, 2R1W/cy. Curve: BN254N, Algo: O-Ate.

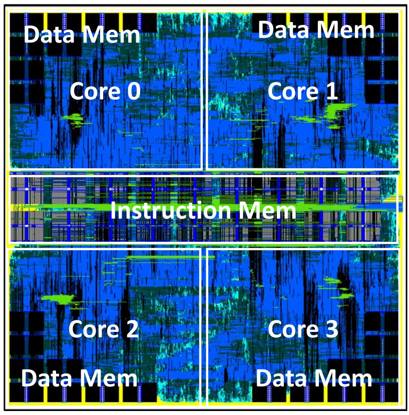
4.5 How agile and practical is the Finesse design framework?

The Finesse design framework offers both agility and practicality. By automating a large part of the design cycle and supporting a wide range of configurations, Finesse allows users to swiftly adapt to diverse application needs. Finesse's effectiveness is best illustrated through various use case scenarios and security considerations.

For Pairing Researchers. Finesse's basic operator kit enables rapid porting of pairing algorithms, allowing users to quickly experiment with new approaches to pairing constructions or novel families of curves, receiving architectural feedback in just minutes. This capability significantly shortens the design cycle, enabling the swift creation of well-optimized hardware accelerators tailored to innovative ideas in pairing-based cryptography.

For Hardware Accelerator Designers. Finesse provides ready-for-use implementations of hardware accelerators in SystemVerilog out of the box, supports generating information aiding RTL-level behavioral simulation. Designers are free to extend and experiment with more advanced ALUs and storage blocks using Finesse, given that it can be incorporated into the hardware abstraction system of Finesse. Finesse's default implementation utilizes standard technology cells, supported by a broad range of EDA toolchains and platforms. Finesse-generated designs are compatible with standard EDA flows, including synthesis, layout, and validation. Figure 12 gives an experimental ASIC layout of quad-core accelerator designed by Finesse, showcasing the practicality and effectiveness of the final product. The framework allows for rapid adjustments to the hardware architecture, further enhancing design agility.

Security Considerations. Finesse provides support at the level of IP cores, meaning it is typically integrated with other user-defined peripheral IPs to form a complete IC die or be deployed on an FPGA board. The physical security of the chip, whether ASIC or FPGA, largely depends on its specific deployment setup. A rigorous and comprehensive evaluation of this aspect is beyond the scope of this paper, which focuses on architecture and design automation.



Technology	40nm LP
Typical Voltage	1.1V
Area	7.992mm²
Gate Count (logic only)	3558.9k NAND2 (equiv.)
SRAM Size	272KiB
Frequency	833MHz
Pairing Curve	BN254N
Pairing Delay	76.3us
Pairing Throughput	52.4kops

Figure 12: Quad-core Finesse chip layout. Note that timing result is slightly better than synthesis results.

Nevertheless, at the current level of support, we can qualitatively assess its resilience against classic attacks, such as basic side-channel and fault-injection attacks. By design, Finesse is inherently resistant to timing attacks, as pairing computations are designed to complete in a fixed number of cycles. Additionally, its instr/data fetch patterns are independent of sensitive inputs, providing a solid basis against attacks that exploit data access patterns. Regarding fault-injection attacks, a bit-flip in the program counter could potentially leak low-rank information about sensitive data. However, this risk can be mitigated by introducing redundancy and/or error correction mechanisms in key memory modules within Finesse.

In conclusion, these use case scenarios and security considerations collectively demonstrate how the Finesse design framework combines agility with practicality, streamlining the design process from algorithm development to final layout and ensuring responsiveness to the evolving demands of pairing-based cryptography.

5 Future Works

Next steps for design space exploration. Our abstraction system supports different memory organization schemes (such as bank configurations) under VLIW. Once hardware support for VLIW is implemented (which is essentially an engineering task), its performance data can be incorporated into the DSE cycle. The choice of memory bank partitioning schemes impacts the memory area at the hardware level, while the number of memory banks affects bank conflict rates during software compilation. This addition introduces a new dimension to the design space, which drives us to pursue more efficient searching strategies. In fact, our parameter space exhibits well-defined adjacency (e.g. adjacent nodes on hypercube), which is suitable for advanced strategies (e.g. simulated annealing).

Constructing a GEM5 model to enhance framework efficiency. In the future work, we intend to develop a equivalent model utilizing GEM5 to improve the efficiency of the Finesse framework. As an open-source system emulator, GEM5 provides a comprehensive simulation of hardware behavior and performance evaluation, enabling us to accurately evaluate hardware performance and power consumption during the design phase. Meanwhile, our open sourced Finesse framework will be upgraded to be compatible with the equivalent GEM5 model.

Supporting wider range of cryptographic algorithms. The hierarchical abstraction of the Finesse enables us to agilely extend the framework to support broader range of classical cryptographic algorithms. For instance, to implement a block cipher algorithm such as AES, it is sufficient to integrate fixed-length octet-stream data operation instructions and type conversion instructions between the $\mathbb{F}_p$ and octet-stream into the ISA.

6 Conclusion

Finesse is an agile design framework for pairing-based cryptography accelerators, providing a novel abstraction that supports automated exploration across multiple layers, spanning algorithms, operators, programs, and hardware, as well as facilitating cross-layer co-design. Through effective co-design, Finesse's accelerator significantly outperforms state-of-the-art solutions in both performance and area efficiency. The agility of Finesse is reflected in its flexible configuration, automated support for both process and optimization, streamlined multi-disciplinary collaboration, and efficient facilitation of the design process for pairing accelerators. By allowing researchers from the algorithm, compiler, and hardware domains to focus on their specific expertise without needing to understand the complete system, Finesse significantly reduces learning and development barriers, making it an effective approach for cryptographic accelerator design.

References

[1] Dan Boneh and Matt Franklin. 2001. Identity-based encryption from the Weil pairing. In *Proceedings of International Cryptology Conference (CRYPTO)*. 213–229. https://doi.org/10.1007/3-540-44647-8_13

[2] Amit Sahai and Brent Waters. 2005. Fuzzy identity-based encryption. In *Proceedings of International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 457–473. https://doi.org/10.1007/11426639_27

[3] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* 17 (2004), 297–319. https://doi.org/10.1007/s00145-004-0314-9

[4] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Proceedings of International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. 177–194. https://doi.org/10.1007/978-3-642-17373-8_11

[5] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[6] Relic-toolkit. 2024. RELIC: a modern research-oriented cryptographic meta-toolkit with emphasis on efficiency and flexibility. https://github.com/relic-toolkit/relic. Accessed: 2024-10-18.

[7] Reza Azarderakhsh, Dieter Fishbein, Gurleen Grewal, Shi Hu, David Jao, Patrick Longa, and Rajeev Verma. 2017. Fast software implementations of bilinear pairings. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 14, 6 (2017), 605–619. https://doi.org/10.1109/TDSC.2015.2507120

[8] Xinyi Hu, Debiao He, Min Luo, Cong Peng, Qi Feng, and Xinyi Huang. 2023. High-Performance Implementation of the Identity-Based Signature Scheme in IEEE P1363 on GPU. *ACM Transactions on Embedded Computing Systems (TAAS)* 22, 2 (2023), 1–35. https://doi.org/10.1145/3564784

[9] Junichi Sakamoto, Daisuke Fujimoto, Riku Anzai, Naoki Yoshida, and Tsutomu Matsumoto. 2024. High-Throughput Bilinear Pairing Processor for Server-Side FPGA Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)* 32, 8 (2024), 1498–1511. https://doi.org/10.1109/TVLSI.2024.3152921

[10] Makoto Ikeda, Tadayuki Ichihashi, and Hiromitsu Awano. 2019. 33us, 94uJ Optimal Ate Pairing Engine on BN Curve over 254b Prime Field in 65nm CMOS FDSOI. In *Proceedings of IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 263–266. https://doi.org/10.1109/A-SSCC47793.2019.9056951

[11] John Hennessy and David Patterson. 2018. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *Proceedings of International Symposium on Computer Architecture (ISCA)*. 27–29. https://doi.org/10.1109/ISCA.2018.00011

[12] Georgios Fotiadis and Elisavet Konstantinou. 2019. TNFS resistant families of pairing-friendly elliptic curves. *Theoretical Computer Science (TCS)* 800 (2019), 73–89. https://doi.org/10.1016/j.tcs.2019.10.017

[13] Razvan Barbulescu, Pierrick Gaudry, and Thorsten Kleinjung. 2015. The tower number field sieve. In *Proceedings of International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. 31–55. https://doi.org/10.1007/978-3-662-48800-3_2

[14] Gabrielle De Micheli, Pierrick Gaudry, and Cécile Pierrot. 2020. Asymptotic complexities of discrete logarithm algorithms in pairing-relevant finite fields. In *International Cryptology Conference (CRYPTO)*. 32–61. https://doi.org/10.1007/978-3-030-56880-1_2

[15] Diego F Aranha, Youssef El Housni, and Aurore Guillevic. 2023. A survey of elliptic curves for proof systems. *Designs, Codes and Cryptography* 91, 11 (2023), 3333–3378. https://doi.org/10.1007/s10623-022-01135-y

[16] A Tengfei Wang, B Wei Guo, and C Jizeng Wei. 2019. Highly-parallel hardware implementation of optimal ate pairing over Barreto-Naehrig curves. *Integration, the VLSI Journal (Integration)* 64 (2019), 13–21. https://doi.org/10.1016/j.vlsi.2018.04.013

[17] Arnab Bag, Debapriya Basu Roy, Sikhar Patranabis, and Debdeep Mukhopadhyay. 2022. Flexipair: an automated programmable framework for pairing cryptosystems. *IEEE Transactions on Computers (TC)* 71, 3 (2022), 506–519. https://doi.org/10.1109/TC.2021.3058345

[18] Oussama Azzouzi, Mohamed Anane, Mouloud Koudil, Mohamed Issad, and Yassine Himeur. 2024. Novel area-efficient and flexible architectures for optimal Ate pairing on FPGA. *The Journal of Supercomputing (TJSC)* 80, 2 (2024), 2633–2659. https://doi.org/10.1007/s11227-023-05578-5

[19] Frederik Vercauteren. 2010. Optimal pairings. *IEEE Transactions on Information Theory (TIT)* 56, 1 (2010), 455–461. https://doi.org/10.1109/TIT.2009.2034881

[20] David Freeman, Michael Scott, and Edlyn Teske. 2010. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology* 23 (2010), 224–280. https://doi.org/10.1007/s00145-009-9048-z

[21] Paulo SLM Barreto, Hae Y Kim, Ben Lynn, and Michael Scott. 2002. Efficient algorithms for pairing-based cryptosystems. In *Proceedings of International Cryptology Conference (CRYPTO)*. 354–369. https://doi.org/10.1007/3-540-45708-9_23

[22] Augusto Jun Devegili, Colm O´hEigertaigh, Michael Scott, and Ricardo Dahab. 2006. Multiplication and squaring on pairing-friendly fields. *Cryptology ePrint Archive* (2006).

[23] Daniel J Bernstein and Tanja Lange. 2007. Faster addition and doubling on elliptic curves. In *Proceedings of International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. 29–50. https://doi.org/10.1007/978-3-540-76900-2_3

[24] Augusto Jun Devegili, Michael Scott, and Ricardo Dahab. 2007. Implementing cryptographic pairings over Barreto-Naehrig curves. In *Proceedings of International Conference on Pairing-Based Cryptography (Pairing)*. 197–207. https://doi.org/10.1007/978-3-540-73489-5_10

[25] Laura Fuentes-Castaneda, Edward Knapp, and Francisco Rodríguez-Henríquez. 2011. Faster hashing to $\mathbb{G}_2$. In *Proceedings of Selected Areas in Cryptography (SAC)*. 412–430. https://doi.org/10.1007/978-3-642-28496-0_25

[26] Yujun Xie, Bin Wang, Lijun Zhang, Xin Zheng, Xiaoling Lin, Xiaoming Xiong, and Yuan Liu. 2022. A high-performance processor for optimal ate pairing computation over Barreto–Naehrig curves. *IET Circuits, Devices & Systems* 16, 5 (2022), 427–436. https://doi.org/10.1049/cds2.12116

[27] Shigeo Mitsunari. 2024. MCL: a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl. Accessed: 2024-10-18.

[28] MIRACL. 2024. MIRACL: Multiprecision Integer and Rational Arithmetic Cryptographic Library. https://github.com/miracl/MIRACL. Accessed: 2024-10-18.

[29] Jaewook Chung and M Anwar Hasan. 2007. Asymmetric squaring formulae. In *Proceedings of IEEE Symposium on Computer Arithmetic (ARITH)*. 113–122. https://doi.org/10.1109/ARITH.2007.11

[30] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal (Integration)* 58 (2017), 74–81. https://doi.org/10.1016/j.vlsi.2017.02.002

[31] Xiaofeng Wang, Peng Zheng, and Qianqian Xing. 2023. Security Analysis of Pairing-based Cryptography. *arXiv preprint arXiv:2309.04693* (2023). https://doi.org/10.48550/arXiv.2309.04693

[32] Razvan Barbulescu and Sylvain Duquesne. 2019. Updating key size estimations for pairings. *Journal of Cryptology* 32 (2019), 1298–1336. https://doi.org/10.1007/s00145-018-9280

[33] Taechan Kim and Razvan Barbulescu. 2016. Extended tower number field sieve: A new complexity for the medium prime case. In *Proceedings of International Cryptology Conference (CRYPTO)*. 543–571. https://doi.org/10.1007/978-3-662-53018-4_20