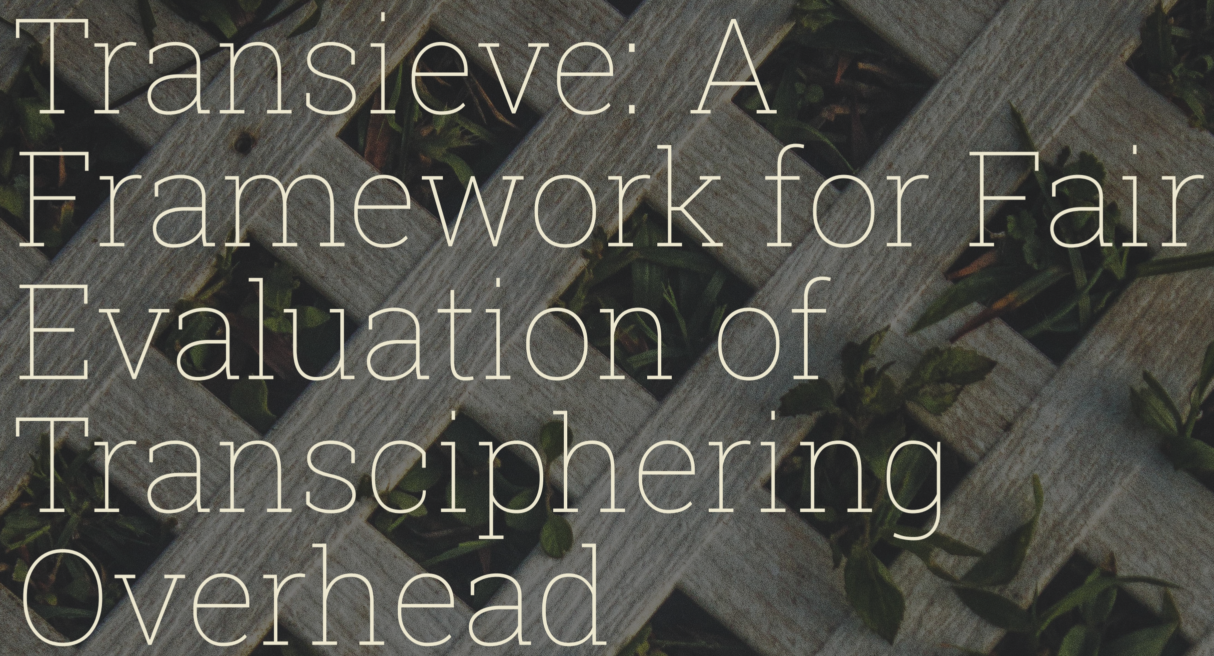


Transieve: A Framework for Fair Evaluation of Transciphering Overhead

A Case Study in Threshold Private Set Intersection

MSc Thesis

Roderick Ras

TU Delft

Transieve: A Framework for Fair Evaluation of Transciphering Overhead

A Case Study in Threshold Private Set Intersection

by

Roderick Ras

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 9, 2026 at 1:30 PM.

Student number:	4295471	
Project duration:	November 2025 – July 2026	
Thesis committee:	Dr. Z. Erkin,	TU Delft, supervisor
	Dr. J. Pouwelse,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

My interest in cryptography began only a few years ago. Before then, I would never have imagined that I would complete a Master's programme in this field, or perhaps any Master's programme at all. Even less did I expect that I would have the opportunity to meet and speak with researchers working at the cutting edge of cryptography. Much of this was made possible by my thesis adviser, Dr. Zeki Erkin, whose enthusiasm and passion for the field played an important role in my decision to pursue this thesis topic. I am deeply grateful for his guidance, encouragement, and trust in me throughout this process, and I will carry his passion for cryptography forward.

I am also grateful to my peers, whose ideas, discussions, and encouragement made this process both more enjoyable and more meaningful. Seeing their progress throughout this journey was a constant source of motivation for my own work. I am also thankful to my colleagues at work for their flexibility, understanding, and support while I was working on this thesis. Their willingness to accommodate me during this period made it much easier to balance my work and research.

Finally, I would like to thank my family and friends for everything. To my mother and father, thank you for your encouragement, support, and belief in me in every endeavour I choose to undertake. Sin Yee, thank you for your patience and understanding, and for always being there. Your support has been invaluable to me. To my brothers and my friends, I appreciate your understanding while I have been less available. Thank you for the encouragement, the conversations, and for patiently enduring my cryptography tangents.

Roderick Ras
Delft, July 2026

Abstract

Fully homomorphic encryption (FHE) lets a server compute on encrypted data without ever decrypting it, giving stronger privacy guarantees when computation is outsourced to infrastructure the data owner does not control. Two costs hold back its use: encrypting data under FHE causes ciphertext expansion by orders of magnitude, and computing on ciphertext is far slower than on plaintext. Transciphering, or hybrid homomorphic encryption, addresses the first cost. Each party uploads its data under a symmetric cipher together with its symmetric key encrypted once under FHE, and the server homomorphically evaluates the cipher's decryption to recover FHE ciphertexts. This trades upload bandwidth for extra server work. How well it pays off depends on the symmetric cipher, the FHE backend it is transciphered into, and how well that backend fits the workload.

To weigh those factors fairly and repeatably, this thesis builds *Transieve*, an evaluation framework that separates a transciphering pipeline into three independent layers: the application workload, the symmetric cipher, and the FHE backend. Each layer can be swapped without disturbing the others, so new workloads, ciphers, and backends can be added and measured on a common footing. The framework is instantiated on threshold private set intersection (TPSI) as its case-study workload, in which items appearing in at least T of N parties' sets are recovered without any party revealing its data to the provider. Each party encodes its set as a Bloom filter, and the server homomorphically evaluates a threshold circuit over the encrypted filters. On this workload, a direct-FHE baseline and a transciphering alternative are built across three FHE backends, TFHE-Boolean, TFHE-shortint, and BFV, each paired with a cipher: the standardised AES-128, the stream ciphers Kreyvium and Grain-128a, the algebraic cipher Rasta, and the prime-field cipher Pasta. All pipelines are compared at a common 128-bit security level on server compute, upload bandwidth and ciphertext expansion, latency and throughput, and recovery accuracy.

The comparison confirms that transciphering delivers its bandwidth saving. At the largest set size, a party's plaintext Bloom filter is about 3.4 MiB. Direct Boolean FHE inflates the upload to about 87 GiB, whereas transciphering keeps it at about 3.8 MiB, close to the plaintext size and a saving of more than four orders of magnitude over direct FHE. This saving is paid for in server work, where no single pipeline is best on every metric. The per-bit cost of homomorphic decryption spans nearly four orders of magnitude across the ciphers: Kreyvium on the Boolean backend is the bandwidth-minimal choice at roughly 310 milliseconds per output bit, while Pasta on BFV is the fastest at roughly 4 milliseconds per bit, packing many bits into prime-field SIMD slots at the cost of a larger upload and a ceiling on the number of parties it can securely support.

The governing factor is how well a cipher's structure matches the backend. The stream ciphers suit the bit-oriented Boolean backend, AES serves as an expensive standardised anchor, and the XOR-dense algebraic cipher Rasta is a negative result on both TFHE backends: the shortint backend removes the bootstrap cost of its dense linear layers, yet only roughly halves its per-bit cost, leaving it uncompetitive, while Pasta matches the prime-field arithmetic of BFV. As the party count grows, the threshold circuit grows combinatorially: on the Boolean backend it overtakes the cipher to become the dominant cost, while BFV evaluates it far more cheaply through SIMD packing and instead meets the security ceiling noted above. The outcome is therefore a situational trade-off rather than a single winner: bandwidth-constrained settings favour the Boolean stream-cipher pipeline, while compute-constrained settings with sufficient memory and a modest party count favour the prime-field BFV pipeline.

Contents

Preface i

Abstract ii

Nomenclature ix

1 Introduction 1
- 1.1 Outsourcing Computation 1
- 1.2 Fully Homomorphic Encryption and Its Limitations 2
- 1.3 Hybrid Homomorphic Encryption and Transciphering 3
- 1.4 Comparing Pipelines for Threshold PSI 4
- 1.5 Contributions 5
- 1.6 Thesis Outline 5

2 Cryptographic Foundations 7
- 2.1 Homomorphic Encryption 7
  - 2.1.1 From Partial to Fully Homomorphic Encryption 7
  - 2.1.2 Learning With Errors 8
  - 2.1.3 Noise Growth and Correctness 9
  - 2.1.4 Bootstrapping 10
- 2.2 FHE Backends 10
  - 2.2.1 TFHE: Boolean and shortint 11
  - 2.2.2 BFV: Prime-Field SIMD 11
  - 2.2.3 What the Backend's Domain Decides 11
- 2.3 Symmetric Encryption 12
  - 2.3.1 Block and Stream Ciphers 12
  - 2.3.2 Internal Structure 12
  - 2.3.3 Classical Design Goals 13
  - 2.3.4 FHE-Friendly Design 13
- 2.4 Transciphering 13
  - 2.4.1 The Construction 13
  - 2.4.2 Homomorphic Evaluation of the Decryption Circuit 14

3 Related Work 15
- 3.1 Transciphering and Hybrid Homomorphic Encryption 15
- 3.2 FHE-Friendly Symmetric Cipher Design 16
- 3.3 Private Set Intersection 16
- 3.4 Privacy-Preserving Outsourced Computation 16

I Building Transieve 18

4 Threshold PSI and the Baseline FHE Pipeline 19
- 4.1 The Threshold-PSI Workload 19
- 4.2 System and Threat Model 20
  - 4.2.1 Parties and Architecture 20
  - 4.2.2 Adversary Model 20
  - 4.2.3 Security Goals 20
  - 4.2.4 Assumptions and Scope 20
- 4.3 Bloom-filter Encoding 21
- 4.4 The Threshold SOP Circuit 21
- 4.5 The Direct-FHE Pipeline 22

4.6 Complexity Analysis . . . 22
4.6.1 Communication . . . 22
4.6.2 Computation . . . 23

5 Transciphering Pipelines 24
5.1 Designing Pipelines . . . 24
5.1.1 The Transieve framework . . . 24
5.1.2 Extending the Framework . . . 25
5.1.3 Cipher Selection . . . 25
5.2 The Boolean Pipeline . . . 26
5.3 The Shortint Pipeline . . . 27
5.4 The BFV Pipeline . . . 27
5.4.1 Leveled BFV . . . 28
5.4.2 Pasta and SIMD Evaluation . . . 28
5.4.3 The Security Ceiling . . . 28
5.5 Generalising the Workload . . . 29
5.6 Complexity Analysis . . . 29
5.6.1 Communication . . . 29
5.6.2 Computation . . . 29
5.6.3 Hardware-Independent Cost . . . 29

II Evaluating Transciphering on TPSI 31

6 Experiments and Evaluation 32
6.1 Security Analysis . . . 32
6.2 Experimental Methodology and Metrics . . . 33
6.2.1 Experimental Setup . . . 33
6.2.2 Datasets and Parameters . . . 33
6.2.3 Server Compute . . . 34
6.2.4 Communication . . . 34
6.2.5 Accuracy . . . 34
6.3 Protocol Scaling . . . 35
6.3.1 Scaling with Set Size . . . 35
6.3.2 Scaling with the Threshold . . . 36
6.3.3 Scaling with the Number of Parties . . . 36
6.3.4 Scaling with the Number of Common Items . . . 37
6.4 Pipeline Comparison . . . 38
6.4.1 Per-Bit Transciphering Cost . . . 38
6.4.2 Upload Bandwidth . . . 38
6.4.3 Direct-FHE SOP Baselines . . . 39
6.5 BFV at High N . . . 40
6.5.1 Kreyvium/Boolean . . . 40
6.5.2 Pasta/BFV . . . 40
6.6 Resource Cost . . . 41

7 Discussion and Conclusion 43
7.1 Interpretation of Findings . . . 43
7.1.1 No Single Pipeline Wins . . . 43
7.1.2 Algebra/Backend Match Decides Server Cost . . . 44
7.1.3 The Protocol Dominates at High Party Counts . . . 45
7.1.4 The BFV Ceiling Is a Transciphering Cost . . . 45
7.1.5 Accuracy Is Governed by the Threshold Policy . . . 45
7.2 Limitations . . . 46
7.3 Future Work . . . 47
7.4 Conclusion . . . 48

References 49

A Implementation Parameters and Reproducibility **53**
A.1 FHE backend parameters 53
A.2 Sampled-prefix caps 53
A.3 Hardware and determinism 53

B The Transieve Interfaces **55**
B.1 Workload layer 55
B.2 Transciphering layer 55
B.3 FHE backend layer 56

C Supplementary Pipeline Figures **57**

D Voting Workload Demonstration Results **60**

List of Figures

1.1 Computing on encrypted data. The server never sees a plaintext. 2
1.2 Transciphering trades a large FHE upload for a compact symmetric ciphertext and a single encrypted key. 3

2.1 The noise growth. Additions raise it slightly, multiplications a lot more. Once it crosses the correctness threshold, decryption fails. 9
2.2 Gentry's re-encryption, or, bootstrapping. Homomorphic evaluation of the decryption circuit with the encrypted secret key, 'unlocking' the inside ciphertext. 10
2.3 In Kreyvium, the homomorphic cost is decided by the operations in its keystream generation: a handful of XORs and 3 ANDs per output bit. 14

4.1 The cloud-assisted system model: owners upload encrypted filters, the server computes the threshold result without ever decrypting. 20

5.1 The Transieve framework. Three decoupled layers: workload, transciphering, homomorphic backend 25

6.1 Recovery and false positives as the party set size grows. The true threshold result is recovered in every row, while the false-positive rate falls but the absolute number of false positives grows. 35
6.2 Recovery and false positives in the threshold sweep ($N = 20$, $S = 10^6$, $C = 10$). Low thresholds are too permissive, while false positives collapse around the upper-middle thresholds. 36
6.3 Threshold-SOP size in the threshold sweep. The number of terms peaks near the middle threshold, which is also the region where false positives become usable. 36
6.4 Threshold-SOP size as the number of parties grows with $T = \lceil N/2 \rceil$. The circuit grows combinatorially, making the threshold circuit itself a major cost driver. 37
6.5 Recovery and false positives as the number of parties grows with $T = \lceil N/2 \rceil$. The false-positive rate follows an even/odd sawtooth caused by the threshold ratio. 37
6.6 Recovery and false positives in the common-IP sweep ($N = 20$, $T = 10$, $S = 10^6$). The exact planted result is recovered in every row. The false-positive background stays roughly fixed until the true signal grows large enough to dominate it. 38
6.7 Per-bit homomorphic symmetric-decryption cost by pipeline. Small-input rows under-fill the relevant block or SIMD structure for some ciphers; the steady-state comparison uses the filled rows. 39
6.8 Per-stage latency breakdown for Kreyvium/Boolean across the set-size sweep ($N = 3$, $T = 2$). The homomorphic symmetric-decryption stage dominates the sampled cost; the threshold SOP is a small share at this party count. 40
6.9 Bandwidth saving of transciphering over each pipeline's direct-FHE baseline. The ratio should be read together with the absolute upload size, because a large saving can also come from a very large direct-FHE baseline. 41
6.10 Latency in the Kreyvium/Boolean high-N sweep. The transciphering cost is almost independent of N, while the SOP grows with the threshold circuit. 42
6.11 Latency in the Pasta/BFV high-N sweep. Direct-BFV SOP timings continue to $N = 12$, while the Pasta/BFV transciphering path is only securable to $N = 4$ under the 128-bit BFV parameter constraint. 42

C.1 Kreyvium/Boolean: stage breakdown (left) and latency (right) across the set-size sweep. 57
C.2 Grain-128a/Boolean (512-bit rerun): stage breakdown (left) and latency (right). 58
C.3 AES-128/Boolean: stage breakdown (left) and latency (right). 58

C.4 AES-128/shortint: stage breakdown (left) and latency (right). 58
C.5 Rasta-525-5/shortint: stage breakdown (left) and latency (right). 58
C.6 Pasta-3/BFV: stage breakdown (left) and latency (right). 59
C.7 High-N stage breakdowns: Kreyvium/Boolean (left) and Pasta/BFV (right) over the party sweep, showing the SOP overtaking the cipher-decrypt stage as N grows. 59

List of Tables

5.1 Overview of the ciphers compared. 26
5.2 Specification-derived, hardware-independent cost of each evaluated pipeline: nonlinear operations per output bit and multiplicative depth, taken from the cipher constants in the implementation (Appendix B) 30

6.1 Protocol accuracy in the set-size sweep ($N = 3$, $T = 2$, $C = 10$). 35
6.2 Common-IP sweep ($N = 20$, $T = 10$, $S = 10^6$). 38
6.3 Representative pipeline comparison at $S = 10^6$, $N = 3$, $T = 2$, $C = 10$. The full-filter time estimates refer to the homomorphic symmetric-decryption stage per party, not to the complete pipeline including the threshold SOP. 39

7.1 Choosing a pipeline by deployment setting. The decision rests on identifying the binding resource for the deployment, then reading across the row. 44

A.1 FHE parameters per backend. The TFHE backends use the `tfhe-rs` library parameter sets; the BFV backend uses `fhe.rs` in leveled mode. 53
A.2 Sampled-prefix size per pipeline. In the Pasta-3/BFV high-N sweep the ring degree, and with it the batch, grows with the circuit depth. 54

Nomenclature

Abbreviations

Abbreviation	Definition
AES	Advanced Encryption Standard
BFV	Brakerski–Fan–Vercauteren (FHE scheme)
BGV	Brakerski–Gentry–Vaikuntanathan (FHE scheme)
CKKS	Cheon–Kim–Kim–Song (approximate-number FHE scheme)
CVP	Closest Vector Problem
FHE	Fully Homomorphic Encryption
FPR	False-Positive Rate
HE	Homomorphic Encryption
HHE	Hybrid Homomorphic Encryption
LHE	Leveled Homomorphic Encryption
LWE	Learning With Errors
NLFSR	Non-Linear Feedback Shift Register
PBS	Programmable Bootstrapping
PHE	Partially Homomorphic Encryption
PSI	Private Set Intersection
RLWE	Ring Learning With Errors
SIMD	Single Instruction, Multiple Data
SOP	Sum of Products
SPN	Substitution–Permutation Network
SVP	Shortest Vector Problem
SWHE	Somewhat Homomorphic Encryption
TFHE	Fast Fully Homomorphic Encryption over the Torus
TPSI	Threshold Private Set Intersection

Symbols

Chapter 2 introduces the homomorphic-encryption primitives in their standard cryptographic notation. It locally writes N for the ring degree and k for the module rank. These uses are specific to that chapter. The symbols below are used in the rest of the thesis, in which N is the number of parties and k the symmetric key.

Symbol	Definition
n	Ring (lattice) dimension of an RLWE ciphertext
q	Ciphertext modulus
p	Plaintext modulus
Δ	Scaling factor placing the message in the high-order bits ($\approx q/p$)
$\mathbf{s}$	Secret key
$\mathbf{a}$	Random public vector of an LWE ciphertext
e	Error (noise) term of a ciphertext
m	Plaintext message (e.g. a party's Bloom-filter bits)
k	Symmetric (transciphering) key
$(\mathbf{a}, b)$	An LWE ciphertext
R_q	The ring $\mathbb{Z}_q[x]/(x^n + 1)$ underlying RLWE
λ	Security parameter (here $\lambda = 128$ bits)
F	Ciphertext expansion factor, $\lvert$ciphertext$\rvert/\lvert$plaintext$\rvert$
N	Number of parties
T	Threshold (an item must appear in at least T of the N sets)
S	Size of each party's set
C	Number of common items planted across the sets
M	Length of a Bloom filter, in bits
K	Number of Bloom-filter hash functions

1 Introduction

Modern organisations are increasingly data-driven [1]. Hospitals, banks, and network operators hold detailed records, patient histories, financial transactions, and depend on computing over them to treat patients, flag fraud, and catch attackers. Those same records are often sensitive, and their exposure can cause lasting harm to the people they describe [2]. Protecting them is no longer only good practice but a legal obligation, under frameworks such as the EU General Data Protection Regulation [3] or the US Health Insurance Portability and Accountability Act. At the same time, the machines that store and process this data are less and less owned by the organisations that hold it, having moved onto infrastructure rented from a handful of cloud providers [4]. These trends pull against each other: the data most worth computing on is the data least safe to expose, and the computers now doing the work are no longer ones their owners fully control.

1.1. Outsourcing Computation

Over the past two decades, organisations have steadily moved their data and their computation off their own premises and onto third-party infrastructure. What began as offloading storage and the occasional batch job is now the default: new systems are designed for the cloud from the start, the so-called cloud-native paradigm, and the shift is still accelerating. Worldwide spending on public cloud services is forecast to pass $720 billion in 2025, up from around $600 billion the year before, and to keep climbing [5]. The reasons for this shift are economic and operational. Instead of buying and maintaining their own machines, organisations turn a large up-front hardware cost, or Capital Expenditure, into an elastic, pay-as-you-go one [6], or Operational Expenditure. On-demand cloud computing has become the essential backbone for web hosting and big data analytics alike [4].

That convenience comes at the cost of control. The moment data leaves an organisation's own premises, the provider is technically able to read, copy, or process it, and the owner is left trusting contracts and reputation rather than anything the system itself enforces [7]. The provider is usually assumed to be honest-but-curious: it runs the requested computation faithfully, but may look at whatever passes through it. This is not just a hypothetical concern. In 2019, an attacker exploited a misconfigured firewall on Amazon Web Services to obtain cloud credentials via server-side request forgery, then exfiltrated over 100 million Capital One customer records stored on the provider's infrastructure [2]. The data was encrypted in transit and at rest, however, once it reached the compute layer, it was plaintext, and valid credentials were all that stood between the attacker and the raw records. The shared platform itself can betray that boundary as well. Tenants on the same physical machine have extracted one another's data through timing side channels [8]. In 2021 a flaw in a widely used managed cloud database let any customer read and alter other customers' data [9]. Besides these technical failures, a single insider or credential compromise can expose data the owner never intended to reveal. In regulated domains such as healthcare and finance, where personal data falls under regulations like the EU General Data Protection Regulation [3], handing plaintext to a third party can be not only risky but unlawful.

Traditional cryptography only solves a part of this problem. Encrypting data at rest and in transit keeps

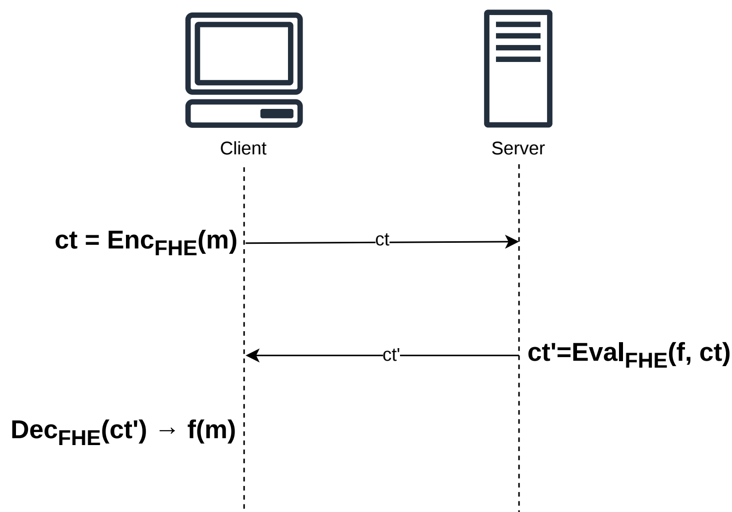


Figure 1.1: Computing on encrypted data. The server never sees a plaintext.

it safe while it sits on disk or travels across the network, but it has to be decrypted before anything useful can be computed on it. At precisely the moment the valuable work happens, the provider holds the plaintext, and the trust assumption we tried to remove reappears [10]. Ideally, we would want to compute directly on the encrypted data, so that the cloud never sees the plaintext at all.

The problem of trust dependency grows when the computation spans several organisations that do not trust one another. The workload that motivates this thesis is collaborative threat intelligence: independent network operators each observe malicious activity, and all of them stand to gain from pooling those observations to identify the indicators of compromise they have in common. Yet none can simply hand over its raw logs, whether for competitive or legal reasons, and there is no trusted party to collect them. The requirement is therefore to hide each party's data from the provider, and ensure the parties learn nothing about each other's inputs beyond the single result they have agreed to compute.

1.2. Fully Homomorphic Encryption and Its Limitations

Computing directly on encrypted data is what fully homomorphic encryption (FHE) makes possible. A server can run an arbitrary computation over ciphertexts. When the owner finally decrypts the output it is identical to what the same computation would have produced on the original plaintext. The data stays encrypted the entire time, from the moment it leaves the owner, through every step on the server, until the owner reads the result. This means that the provider does the work without ever learning what it worked on, which closes the gap that traditional encryption leaves open at the moment of computation (Figure 1.1).

The concept predates its practicality. Gentry gave the first working construction in 2009, settling the question of whether arbitrary computation on ciphertext was possible at all [11]. What followed was a continuous engineering effort: a family of schemes built on the hardness of lattice problems (Shortest Vector Problem, Closest Vector Problem) steadily cut the cost, moving FHE from a theoretical idea to software that runs on modern hardware [12, 13, 14, 15]. It has even reached deployment: Apple ships homomorphic private information retrieval in its operating systems [16], and Microsoft's Edge password monitor checks a user's credentials against known leaks using an FHE-based private set intersection, without revealing them [17].

That progress has not made FHE free, and the cost it still carries is the reason this thesis exists. Two penalties separate computing on ciphertext from computing on plaintext.

Firstly, encrypting data under an FHE scheme inflates it by several orders of magnitude. Only a few bits of plaintext can become kilobytes of ciphertext [18]. When computing on few values this hardly

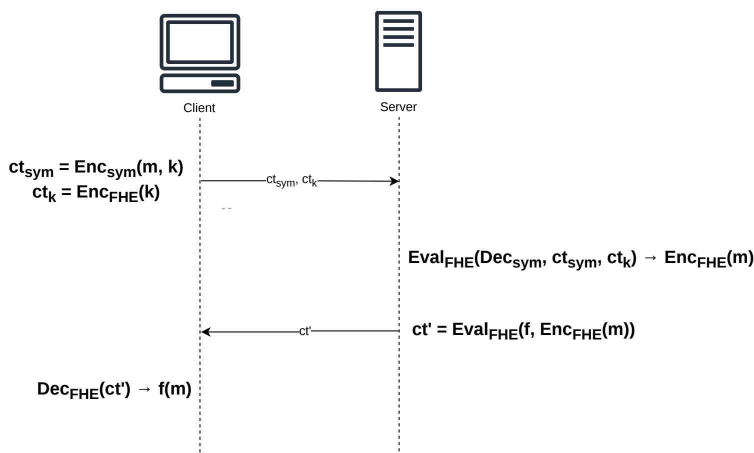


Figure 1.2: Transciphering trades a large FHE upload for a compact symmetric ciphertext and a single encrypted key.

matters, but the datasets that make outsourcing worthwhile are large and there the upload alone can reach gigabytes for a single participant. The bandwidth advantage that drew the work to the cloud in the first place is spent before any computation begins.

Secondly, each operation on a ciphertext is far slower than its plaintext equivalent. As a computation grows the schemes must periodically run an expensive noise-control step called bootstrapping to ensure correctness when decrypting [10, 11]. Work that takes milliseconds in the clear can take minutes or longer under FHE.

These two costs have different characters. The compute cost is an inherent property of working on encrypted data and reducing it is the focus of ongoing research. The upload penalty, however, can be approached at the protocol level and is the focus of this thesis. The technique it uses to achieve this is called transciphering.

1.3. Hybrid Homomorphic Encryption and Transciphering

The upload penalty can be removed without giving up any of FHE's protection by combining two ciphers instead of relying on FHE alone. A lightweight symmetric cipher carries the data to the server compactly, and the FHE scheme computes on it once it is there. This pairing of a symmetric cipher for transport with an FHE scheme for computation is known as *hybrid homomorphic encryption* (HHE). Each party encrypts its data under the symmetric cipher, whose ciphertext is essentially the same size as the plaintext, and uploads it together with its symmetric key encrypted once under FHE. Before the server can compute, it converts that symmetric ciphertext into FHE ciphertext, a step called *transciphering*. The approach was first proposed as a way to lower communication overhead with the cloud [19] and was later made practical [18]. Figure 1.2 shows the transciphering flow.

Holding the FHE-encrypted key, the server homomorphically evaluates the cipher's decryption on the symmetric ciphertext, which yields FHE encryptions of the original data, lifting symmetric ciphertext into homomorphic ciphertext. From this point, the computation proceeds exactly as before. The provider has still seen nothing in the clear. A large upload has become a compact symmetric ciphertext and a single, fixed-size homomorphic key. Additionally, the heavy computation now happens on the server.

The computation cost has been offloaded to the server. It must homomorphically evaluate the cipher's decryption circuit before it can even begin the real computation, which is only as cheap as the cipher allows. A standard block cipher like AES is a poor fit: its round function is built from exactly the operations FHE finds most expensive, so evaluating it homomorphically is slow. The cipher is therefore one of the key factors that decides whether transciphering pays off or not.

This is part of a larger trend in how symmetric cryptography is used. For most of its history a cipher was judged by how fast it ran in hardware or software. Increasingly, ciphers are designed instead to serve as primitives inside larger constructions, where the cost that matters is not traditional CPU or hardware constraints, but homomorphic operations, communication rounds in secure multiparty computation, or constraints in a zero-knowledge proof [20, 21]. Each of these settings measures a cipher by different metrics; multiplicative depth, multiplicative size, and algebraic constraint counts, respectively, and each has produced its own family of designs. Analysing these algebraic trade-offs to select or design the ideal primitive is itself an active line of cipher design.

Choosing the right cipher for FHE transciphering is challenging due to a wide variety of available candidates. The optimal selection depends on two variables: the specific FHE backend handling the transciphering and the nature of the subsequent workload. This thesis addresses this challenge by designing, implementing, and evaluating these factors for a concrete workload.

1.4. Comparing Pipelines for Threshold PSI

The workload this thesis uses is threshold private set intersection (TPSI). Several parties each hold a private set and they want to learn which elements appear in at least T of the N sets, while revealing nothing else. This protocol maps directly onto the collaborative threat intelligence mentioned earlier, where each operator's set is the indicators it has observed and the threshold picks out the indicators seen by enough independent parties to be worth acting on. TPSI is an appropriate test for transciphering because it is bit-oriented, its inputs are large, and it runs across several distrusting parties, so both costs from Section 1.2, upload size and server compute, are simultaneously considered.

The growing body of work on HHE has produced a wide choice of FHE-friendly ciphers and FHE backends to pair them with [22]. Which combination is best for a given workload, measured end to end and on equal terms, is the question that decides a deployment. The deployable application is not a cipher or a backend on its own, but the whole pipeline: an FHE backend, the symmetric cipher transciphered into it, and the evaluation of the threshold computation on top. These three choices are not independent, but form a chain. The workload fixes the shape of the computation, which decides the FHE backend that can carry it out efficiently, and the backend in turn decides which symmetric ciphers can be transciphered into it cheaply, since a cipher is only fast under FHE when it is built from the operations its target scheme finds cheap. A pipeline therefore has to be evaluated as a whole, from the workload down, rather than judged from parts measured in isolation under conditions that do not match. The central question is therefore:

> *Under a fair, end-to-end comparison on a single workload, how do transciphering pipelines, each an FHE backend paired with a symmetric cipher, compare, and what decides which pipeline best fits a given deployment?*

We answer this question for threshold PSI, breaking it into the supporting questions below. All questions are asked on that specific workload, at a common 128-bit security level [23]:

- **RQ1 (server compute).** What does it cost, per output bit, to homomorphically evaluate each backend's best-fit cipher?
- **RQ2 (bandwidth).** How does transciphering's upload saving over direct FHE scale with the size of the data?
- **RQ3 (pipeline comparison).** Across latency, throughput, and bandwidth, is there a single best pipeline or does the best choice depend on the deployment scenario?
- **RQ4 (protocol scaling).** How do compute, bandwidth, and accuracy scale with the number of parties, the threshold, the set size, and the overlap between sets?
- **RQ5 (algebra match).** How much does matching a cipher's algebra to its FHE backend matter for homomorphic cost?
- **RQ6 (accuracy).** How does the Bloom filter's false-positive rate trade off against bandwidth?

Scope. The setting is single-key FHE under an *honest-but-curious* server, the standard assumption that the provider follows the protocol faithfully but may inspect whatever it processes. Malicious adver-

saries and the cryptanalysis of the symmetric ciphers themselves are out of scope. The full security model is set out in Section 4.2.

1.5. Contributions

Transciphering has been studied mostly in pieces. A new FHE-friendly cipher is typically proposed and benchmarked against a single, fixed FHE scheme, and each study reports its own parameters and security level, so the numbers rarely transfer from one paper to the next. A recent systematisation closes much of this, comparing many ciphers and transciphering methods end to end, though on arithmetic, single-party workloads over the SIMD-friendly schemes [24]. What is still missing is that comparison for a bit-valued, multi-party workload across the bit-oriented backends, held at one common security level. This thesis addresses that gap. Rather than proposing yet another cipher, it makes the fair comparison of whole pipelines the object of study, with threshold PSI as a demanding, bit-oriented use case. It makes three contributions:

- ***Transieve*, an open and extensible evaluation framework.** We build the implementation once, as a framework we call *Transieve*[1], that keeps the workload (its encoding and the circuit evaluated under FHE) separate from the transciphering and FHE layers. It is instantiated on three FHE backends, the TFHE Boolean and shortint backends [25] and a prime-field BFV backend [26], across five symmetric ciphers (Kreyvium, Grain, AES, Rasta, and Pasta), each with a direct-FHE baseline. Because the three components connect only through narrow interfaces, a different cipher, backend, or workload can be added and evaluated under identical conditions without touching the rest. A secure voting tally, for example, shares the same threshold structure and is implemented on the same pipelines as an extensibility demonstration, though only threshold PSI is measured. This makes the comparison repeatable and *Transieve* reusable as an evaluation tool beyond this thesis.
- **A fair-comparison methodology for transciphering pipelines.** We define the conditions under which whole pipelines can be measured against one another as deployable units rather than isolated components: a single realistic workload, a common 128-bit security level [23], and one consistent set of end-to-end metrics, namely server compute per output bit, upload bandwidth and ciphertext expansion, latency and throughput, and output accuracy. A per-bit sampling and extrapolation scheme brings the slower FHE paths onto the same measurable basis.
- **The first end-to-end comparison of transciphering for threshold PSI.** Transciphering has been demonstrated and benchmarked for machine-learning inference and statistical workloads, all single-party, but not for private set intersection. Using *Transieve* and the methodology above, we compare the pipelines end to end and report where transciphering helps and where it does not, which cipher fits which backend and why, how the best choice shifts with the deployment, and the limits the pipelines reach, including a backend's security ceiling at scale. The outcome is not a single recommended cipher but comparable evidence about the bandwidth-against-compute trade-off, and a structured way to navigate it for a given workload.

1.6. Thesis Outline

The remainder of this thesis is organised into two parts, preceded by background. Chapter 2 covers the cryptographic foundations: the FHE backends and their cost models, the symmetric ciphers in the comparison, and the transciphering construction. Chapter 3 reviews related work on HHE and on private set intersection, and places this thesis among it.

Part I builds *Transieve*, the evaluation framework the comparison runs on. Chapter 4 describes the threshold PSI workload, the system and threat model under which it is computed, and the baseline pipeline that evaluates it directly under FHE, without transciphering. Chapter 5 adds the transciphering stage, showing how each cipher is evaluated homomorphically and coupled to its backend.

Part II presents the fair comparison. Chapter 6 sets out the evaluation methodology and metrics, then reports the experiments and the cross-pipeline evaluation under identical conditions. Finally, Chapter 7

[1] The implementation is available at `https://github.com/RRas/Transieve`. Appendix A records the parameters needed to reproduce the measurements.

discusses what the results mean, the limitations of the experiment, and directions for future work, and concludes.

2

Cryptographic Foundations

This chapter introduces the cryptographic background the rest of the thesis relies on. Each topic is explained only to the extent that it impacts subsequent design choices, costs, and comparisons. The complete theory is left to the works cited along the way. The chapter begins with fully homomorphic encryption and the two costs that motivate the thesis (Section 2.1). Then it covers the specific homomorphic backends the work runs on (Section 2.2) and the symmetric ciphers as primitives for larger constructions (Section 2.3). Finally, it closes with the HHE construction that combines the two: Transciphering (Section 2.4).

2.1. Homomorphic Encryption

Homomorphic encryption is what enables a server to compute on data it cannot read. This section explains how that is possible and, more importantly for this thesis, where its two costs come from. Both the ciphertext expansion and the slow computation that motivate transciphering trace back to a single source, namely, the noise that every such ciphertext carries [10].

2.1.1. From Partial to Fully Homomorphic Encryption

An encryption scheme is homomorphic when an operation performed on ciphertexts corresponds to an operation on the plaintexts inside them. If a server adds two ciphertexts and the owner decrypts the result, the owner reads the sum of the two original messages, while the server only ever handled encrypted values. Figure 1.1 shows this flow: the data is encrypted once by the owner, processed on the server while encrypted, and decrypted only at the end, so the plaintext never leaves the owner's control.

Homomorphic schemes differ in how much they can compute. *Partially* homomorphic schemes support one operation without limit, such as the multiplicative homomorphism of textbook RSA [27] or the additive homomorphism of Paillier [28]. *Somewhat* homomorphic schemes support both addition and multiplication but only for a bounded number of operations. A *leveled* scheme fixes that bound in advance and chooses its parameters to fit a circuit of known depth. A *fully* homomorphic scheme removes the bound entirely and can evaluate arbitrary circuits. Craig Gentry gave the first fully homomorphic construction in 2009, settling the question of whether unbounded computation on ciphertext was possible at all [11].

The computations this thesis evaluates under encryption, the threshold circuit of Chapter 4 and the symmetric decryption circuits of Chapter 5, mix many additions and multiplications. This means a partially homomorphic scheme is not suitable. The backends *Transieve* uses handle the resulting noise in two different ways, explained in Section 2.2: the TFHE backends bootstrap after every operation, making them fully homomorphic, while the BFV backend runs in leveled mode, with its multiplicative depth fixed in advance. All rest on the same hardness assumption, namely, Learning With Errors.

2.1.2. Learning With Errors

Modern homomorphic encryption is built on the Learning With Errors (LWE) problem [29], with its hardness coming from lattices. A *lattice* is the set of all integer linear combinations of n linearly independent basis vectors, in essence, a regular grid of points in $\mathbb{R}^n$. Two of its computational problems are believed hard even for quantum computers: the Shortest Vector Problem (SVP), finding the shortest nonzero vector of the lattice, and the Closest Vector Problem (CVP), finding the lattice point nearest a given target. LWE is provably as hard as approximating these worst-case problems [29]. This problem underpins the security of every homomorphic scheme in this thesis.

LWE hides a message inside noisy linear equations. The secret key is a vector $\mathbf{s} \in \mathbb{Z}_q^n$, where q is the ciphertext modulus. To encrypt a message $m \in \mathbb{Z}_p$, with $p < q$ the plaintext modulus, one samples a uniform vector $\mathbf{a} \in \mathbb{Z}_q^n$ and a small error e from a narrow, discrete Gaussian distribution, and sets

$$b \;=\; \langle \mathbf{a}, \mathbf{s} \rangle + e + \Delta\, m \pmod q, \qquad \Delta = \lfloor q/p \rfloor,$$

where Δ scales the message into the high-order bits. The ciphertext is the pair $\mathbf{c} = (\mathbf{a}, b)$. To an attacker without $\mathbf{s}$, the public vectors $\mathbf{a}$ generate a lattice, and b lies close to it, off by only the small error e. Then, recovering $\mathbf{s}$, and with it m, is a closest-vector-type problem. LWE's hardness is supported by reductions from worst-case lattice problems [29]. To the key holder, decryption is simple. They subtract the mask and round off the error,

$$b - \langle \mathbf{a}, \mathbf{s} \rangle = \Delta\, m + e \pmod q, \qquad m = \left\lfloor \tfrac{b - \langle \mathbf{a}, \mathbf{s} \rangle}{\Delta} \right\rceil,$$

which returns the correct m as long as $|e| < \Delta/2$. The error e is both what makes the problem hard and the factor that impacts every cost in this chapter. Because the construction is almost linear in the secret, operating on ciphertexts operates on the messages they hold.

The ciphertext just described is one special case of a more general construction. Both backends in this thesis are instances of *General LWE* (GLWE), which parameterises the same noisy equation over polynomials. A GLWE encryption of a message M under a secret $\mathbf{S} = (S_0, \ldots, S_{k-1})$ is the tuple

$$(A_0, \ldots, A_{k-1},\, B), \qquad B = \sum_{i=0}^{k-1} A_i S_i + E + \Delta\, M,$$

where the message M, the random masks A_i, the error E, and each secret component S_i are all elements of the ring $R_q = \mathbb{Z}_q[X]/(X^N + 1)$. Two parameters fix the shape: the *rank* k, how many polynomials the secret holds, and the ring *degree* N. Their product $n = kN$ is the lattice dimension that sets the security level, and the two backends are the two special cases that leave it unchanged:

- **LWE** ($N = 1$): the polynomials collapse to single integers. The secret is a length-n vector $\mathbf{s}$, the masks a_i and error e are scalars, and B reduces to $b = \langle \mathbf{a}, \mathbf{s} \rangle + e + \Delta m$, exactly the ciphertext above. It carries one value, and is the form the TFHE backends of Section 2.2 compute on directly.
- **RLWE** ($k = 1$) [30]: the secret is a single degree-n polynomial. The mask, error, and message are all ring elements, and the inner product $\sum_i A_i S_i$ becomes one polynomial multiplication. This is the form the BFV backend uses.

Working in a ring rather than with length-n vectors shrinks keys and ciphertexts, speeds up the arithmetic through the number-theoretic transform, and adds a property the thesis relies on later. By the Chinese Remainder Theorem the plaintext space splits into many independent *slots*, so a single RLWE ciphertext carries an entire vector of values and one homomorphic operation acts on all of them at once. This is the Single Instruction, Multiple Data (SIMD) batching that the BFV backend of Section 2.2 exploits.

These two forms directly expose the origin of homomorphic encryption's first overhead, ciphertext expansion, and demonstrate that they do not pay this penalty equally. A ciphertext is a pair over $\mathbb{Z}_q$ (or R_q), so it occupies on the order of $n \log_2 q$ bits, with n in the thousands and $\log_2 q$ in the hundreds. In the LWE form all of those bits carry a single value, so even one plaintext bit becomes thousands of bits of ciphertext. Packing into RLWE slots is the first mitigation. The same ciphertext then holds an entire vector, so its size is amortised across the slots and the expansion per value falls accordingly.

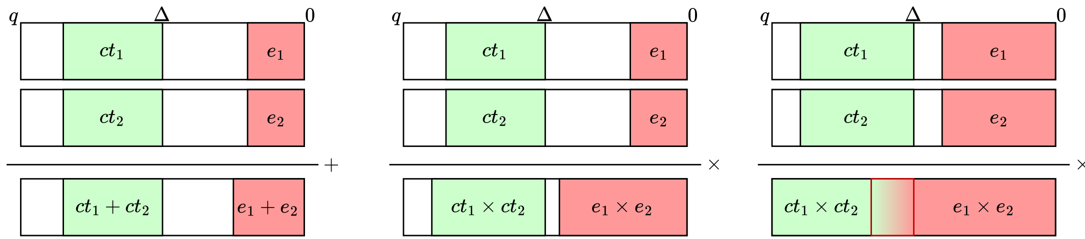


Figure 2.1: The noise growth. Additions raise it slightly, multiplications a lot more. Once it crosses the correctness threshold, decryption fails.

Even so, the ciphertext expansion factor $F = |\text{ciphertext}|/|\text{plaintext}|$ stays at least around ten even for densely packed ciphertexts [18, 19]. This thesis touches upon both ends of the range. The BFV backend packs each party's bits into SIMD slots and so sits near that lower bound, whereas the bit-oriented TFHE backends encrypt one bit per ciphertext, with no batching across the data, and their expansion climbs to several orders of magnitude. This ciphertext expansion is the first of the two costs and is the reason transciphering was proposed in the first place [19].

2.1.3. Noise Growth and Correctness

Decryption is correct only while the error stays small. From the previous subsection, a ciphertext decrypts to the right message precisely when $|e| < \Delta/2$, that is, while the error has not grown past half the gap $\Delta \approx q/p$ that separates encoded messages. A freshly encrypted ciphertext sits well inside this margin. Homomorphic operations, however, eat into this space, but each in a different way.

Addition is exact and cheap. Given ciphertexts $\mathbf{c}_1 = (\mathbf{a}_1, b_1)$ and $\mathbf{c}_2 = (\mathbf{a}_2, b_2)$ encrypting m_1 and m_2 under the same key, their componentwise sum $\mathbf{c}_+ = (\mathbf{a}_1 + \mathbf{a}_2,\, b_1 + b_2)$ satisfies

$$(b_1 + b_2) - \langle \mathbf{a}_1 + \mathbf{a}_2, \mathbf{s} \rangle = \Delta\,(m_1 + m_2) + (e_1 + e_2) \pmod q,$$

so it decrypts to $m_1 + m_2$ with error $e_1 + e_2$. The errors simply add, $|e_+| \le |e_1| + |e_2|$, so addition grows the noise only linearly.

Multiplication is what impacts the cost significantly, and the two scheme families handle it differently. In a leveled scheme such as the BFV backend used here, producing a ciphertext that decrypts to $m_1 m_2$ requires combining the two ciphertexts in a way that is quadratic in the secret, a tensor product, followed by a *relinearisation* step that rewrites the result as an ordinary ciphertext under $\mathbf{s}$ [12, 13]. The messages multiply as intended, but so do the errors. The new error behaves like

$$|e_\times| \;\lesssim\; \delta \cdot \big(|e_1| + |e_2|\big),$$

where the expansion factor $\delta \gg 1$ is fixed by the plaintext modulus and the ring dimension. Multiplication *multiplies* the noise rather than adding it, and this asymmetry is what bounds computation. Along a chain of L successive multiplications the error grows on the order of δ^L, so the correctness condition $|e| < \Delta/2$ holds only while

$$L \;\lesssim\; \frac{\log(q/p)}{\log \delta}.$$

The length of the longest such chain is the circuit's *multiplicative depth*, and this longest chain is what a ciphertext's finite *noise budget* can afford. Figure 2.1 shows the budget being spent until the error crosses the threshold beyond which decryption fails.

The same bound exposes a second pressure on the first cost. Raising the ciphertext modulus q buys more depth, since the budget $\log(q/p)/\log\delta$ grows with q, but it also widens every coefficient and inflates the $n \log_2 q$ ciphertext size, so depth is traded directly against ciphertext expansion. A leveled scheme such as BFV makes that trade once: it fixes the depth in advance, sizes q to just fit, and never exceeds it [12, 13]. That is efficient when the depth is known and quite shallow. However, the transciphering pipeline must evaluate a full cipher decryption and a threshold circuit on top, so going beyond a fixed depth cannot be avoided.

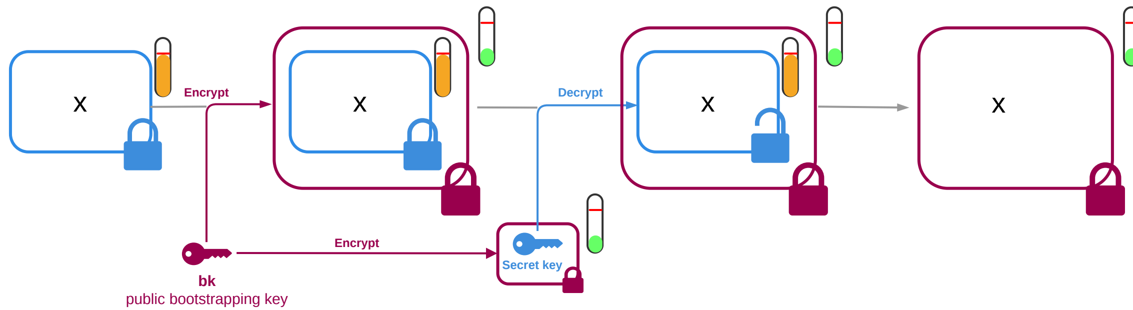


Figure 2.2: Gentry's re-encryption, or, bootstrapping. Homomorphic evaluation of the decryption circuit with the encrypted secret key, 'unlocking' the inside ciphertext.

The TFHE backends answer the noise problem in another way. Rather than tensoring and accepting the growth above, they evaluate a ciphertext-by-ciphertext product, like Boolean AND, with a single *bootstrap*: one operation that computes the gate and, at the same time, resets the error to a fixed low level. Because the noise is refreshed at every such step it never accumulates, so there is no δ^L growth and no depth bound. The price is then exactly one bootstrap per operation. The next subsection explains what a bootstrap is, and Section 2.2 shows how each backend builds on one of these two answers.

2.1.4. Bootstrapping

Bootstrapping is the operation that removes the depth bound and makes a scheme fully rather than somewhat homomorphic. It takes a ciphertext whose noise budget is nearly spent and produces a fresh ciphertext of the same message with the budget restored (Figure 2.2). It does this by homomorphically evaluating the scheme's own decryption function: given a public *evaluation key* (or *bootstrapping key*) consisting of an encryption $\mathrm{Enc}(\mathbf{s})$ of the secret key, the server computes $\mathrm{Enc}\big(\mathrm{Dec}_{\mathbf{s}}(\mathbf{c})\big)$. This yields a fresh encryption of the same message m whose error is reset to the small level produced by the decryption circuit, all without ever exposing the plaintext or the key. This was Gentry's central idea and the step that turned somewhat homomorphic encryption into a fully homomorphic scheme [11].

However, bootstrapping is expensive, and running it often is what makes FHE slow. This is the second of the two costs. Modern schemes approach this bottleneck differently: large-modulus schemes like BFV amortise the cost by bootstrapping entire vectors of slots at once, whereas the TFHE family minimises the price of a single bootstrap. In its *programmable* form, TFHE even evaluates a table lookup simultaneously, making bootstrapping cheap enough to run after every elementary gate operation [15]. Section 2.2 returns to this, because how a backend bootstraps is exactly what fixes its per-operation cost and, through that, dictates which symmetric ciphers it can transcipher efficiently.

Both costs introduced in this chapter trace back to the noisy lattice ciphertext: its $n \log_2 q$ structure is what inflates its size (the expansion cost), while keeping the noise in check as the circuit grows is what forces the periodic bootstrapping that dominates the compute cost. Transciphering, the subject of Section 2.4, directly attacks the expansion problem: the client uploads a compact symmetric ciphertext instead of FHE ciphertexts, and the conversion into FHE form moves to the server, which performs it by homomorphically evaluating the symmetric decryption circuit. This shift, however, alters rather than eliminates the computational burden. The server must now homomorphically evaluate a symmetric decryption circuit before processing any data. Transciphering therefore introduces a new baseline computational overhead, the severity of which depends entirely on selecting an FHE-friendly cipher optimised for the specific backend's bootstrapping paradigm.

2.2. FHE Backends

Section 2.1 was an overview of homomorphic encryption. The comparison in this thesis runs on three concrete backends drawn from two scheme families: the Boolean and shortint backends of TFHE, and a BFV backend.

2.2.1. TFHE: Boolean and shortint

TFHE takes the opposite stance to the leveled schemes mentioned in Section 2.1. Instead of avoiding bootstrapping because it is expensive, it makes a fast bootstrap the basic operation [15]. Its *programmable bootstrapping* (PBS) does two things at once. It refreshes a ciphertext's noise and it evaluates a small lookup table on the encrypted value in the process. A PBS internally combines a blind rotation with a key-switch, but for cost purposes it is one indivisible unit of work whose price is essentially fixed, independent of the table it applies. Using this constant cost, we can directly reason about the computational overhead when transciphering with TFHE.

The two TFHE backends differ in what a ciphertext holds.

Boolean packs a single bit per ciphertext. Every binary gate, AND, OR, XOR, and the rest, is evaluated by one PBS whose lookup table is the gate's truth table, the original *gate bootstrapping* mode of TFHE. A Boolean circuit's homomorphic cost is therefore its gate count: every gate is one bootstrap. Importantly, this count includes XOR, which is not free here as it would be in a leveled scheme. This is relevant for the comparison, because a cipher that looks cheap by its multiplication count can still be expensive if it is dense in XORs (Chapter 6 returns to this). The domain is bit-exact, which fits bit-oriented ciphers and the bit-valued threshold computation of the workload described in Chapter 4 directly.

shortint packs a small integer per ciphertext, split into a *message* part and a *carry* part of a few bits each. A PBS here evaluates an arbitrary lookup table over that small integer. This is the more general *programmable* bootstrapping of which gate bootstrapping is a special case. The carry space lets several additions accumulate before a bootstrap is needed to clear it, so addition becomes a *leveled* operation with no per-operation bootstrap, while nonlinear functions, like AND operations, still cost a PBS. Because an XOR of bits can be carried out as such an addition, shortint makes XOR cheap and incurs costs mainly for the nonlinear work. This leveled-XOR trick is what Trivium-on-TFHE exploits [31], and it is why a cipher whose cost is dominated by XOR is far cheaper on shortint than on Boolean, while staying in the same TFHE scheme.

Within one scheme family, then, Boolean charges for every gate while shortint charges mostly for the nonlinear ones, at the cost of larger ciphertexts and a slower individual bootstrap. Which trade wins depends on the cipher, which is the comparison's point.

2.2.2. BFV: Prime-Field SIMD

BFV is a leveled RLWE scheme that works over a prime field $\mathbb{F}_p$, with $p = 65537$ here [13]. It supports bootstrapping, but this thesis will only make use of leveled BFV. The scheme thus fixes a multiplicative-depth budget in advance and sizes its modulus chain to fit. Two properties set it apart from the TFHE backends.

The first is SIMD batching. As noted for RLWE in Section 2.1, the ring structure packs many independent $\mathbb{F}_p$ values into the slots of one ciphertext allowing a single homomorphic operation to act on all of them at once. A computation that is identical across many data values runs once rather than once per value, which is what makes BFV fast on wide, data-parallel workloads.

The second is that the cost is multiplicative, not per-gate. Additions and multiplications by public constants are cheap and barely touch the depth budget. Only ciphertext-by-ciphertext multiplications consume it. The limiting factor is therefore multiplicative depth, so the cipher that suits BFV is one whose nonlinearity is low-degree field arithmetic rather than bit logic (Chapter 5).

The depth budget is also bounded, and that bound is a result this thesis reports rather than just a caveat. Because $p = 65537$ caps the SIMD ring degree, and the degree in turn caps the modulus that is still secure at $\lambda = 128$ bits [23], a circuit that is too deep cannot be evaluated at the target security level. Chapter 6 shows the high-party-count threshold runs reaching this ceiling in the TPSI workload use case.

2.2.3. What the Backend's Domain Decides

The three backends span three domains: a single bit (TFHE Boolean), a small integer with message and carry slots (TFHE shortint), and a vector of prime-field words (BFV). The choice of domain decides which operations are cheap, and through that, which symmetric cipher can be transciphered into the

backend without too high a cost:

- a bit-oriented cipher with few nonlinear gates suits *Boolean*;
- a bit-oriented cipher that is XOR-heavy but light on nonlinear gates suits *shortint*, where XOR is leveled;
- a cipher whose nonlinearity is low-degree $\mathbb{F}_p$ arithmetic suits *BFV*, with its SIMD slots and depth budget.

No single backend is best for every cipher, and no single cipher is best for every backend. This is why the comparison in this thesis is across combinations, a backend and its best-fit cipher together, rather than across ciphers or backends in isolation. The workload enters as well: a bit-valued workload such as threshold PSI is native to the TFHE backends but must be encoded into $\mathbb{F}_p$ words to run on BFV. Section 2.4 and Part I make these couplings concrete.

2.3. Symmetric Encryption

Symmetric encryption is the other half of transciphering. It carries each party's data to the server compactly, after which the server has to undo it homomorphically. Section 2.2 fixed the cost of doing things homomorphically, so the question this section builds toward is not how fast a cipher runs on a processor but how expensive its operations are once they have to be evaluated under FHE. Choices made in symmetric cipher design differ vastly when taking these costs into account.

2.3.1. Block and Stream Ciphers

A symmetric cipher uses a single secret key for both encryption and decryption, in contrast to the asymmetric, public-key structure of FHE. Two families matter here.

A *block cipher* transforms fixed-size blocks of data under a keyed, invertible permutation. AES is the standard example, which works on 128-bit blocks [32]. Its decryption is the inverse permutation, which is structurally different from encryption.

A *stream cipher* instead generates a pseudo-random keystream from the key and a public initialisation value, then encrypts by combining the keystream with the data using XOR. Trivium [33] and its 128-bit-security successor Kreyvium [18], along with Grain [34], are examples used later in this thesis. For a stream cipher, encryption and decryption are the same: a simple XOR operation.

Importantly, for transciphering, it is the *decryption* circuit that matters, because that is what the server evaluates homomorphically. For a stream cipher this reduces to generating the keystream homomorphically and then XOR-ing, so the whole homomorphic cost sits in keystream generation. For a block cipher it is the inverse permutation. In any case, the cost of transciphering a cipher is the cost of that circuit under the backend's cost model from Section 2.2.

2.3.2. Internal Structure

Most symmetric ciphers are built by repeating a round several times, and each round interleaves two kinds of layer: a linear and a nonlinear layer. A *linear* (or affine) layer permutes and mixes bits to spread local changes across the whole state. However, a linear layer alone is insufficient for security. Without a nonlinear component, the entire cipher collapses into a single system of linear equations. An attacker could then trivially break the cipher via basic algebraic attacks, such as Gaussian elimination, or exploit it using linear and differential cryptanalysis, completely bypassing the diffusion provided by the permutations. A *nonlinear* layer is therefore required to disrupt these linear relationships, providing the cryptographic strength necessary to resist linear and differential cryptanalysis. Security comes from stacking enough rounds of the two layers to ensure that any algebraic description of the cipher becomes too complex to solve.

Block ciphers come in two shapes. In a *substitution-permutation network* (SPN), each round applies a nonlinear substitution to the whole state and then a linear mixing step. AES is an example of an SPN. In a *Feistel network*, the state is split and a round function is applied to one half and combined with the other.

The nonlinear layer is where the families differ most, and it is what decides homomorphic cost. It

appears in a few forms: a table-based substitution box, such as the finite-field inversion in the AES S-box; a bit-sliced logical map, such as the Keccak χ function used by Rasta [35]; a handful of AND products between state bits, as in the Trivium and Kreyvium stream ciphers (Figure 2.3); or a low-degree power map over a prime field, such as cubing. These cost different amounts under FHE, which is further explained in Section 2.3.4 and Part I. The number of rounds and the key schedule round out the design: more rounds buy more security margin, but under FHE they also buy more homomorphic work.

2.3.3. Classical Design Goals

A symmetric cipher is traditionally judged by three principles. The first is Shannon's pair of principles, *confusion* and *diffusion*: each ciphertext bit should depend on the key in a complicated way, and each plaintext bit should influence many ciphertext bits [36]. The nonlinear and linear layers of the previous subsection are what realise them. The second is resistance to the standard cryptanalytic families, mainly differential [37] and linear [38] cryptanalysis, but also, for algebraically simple designs, algebraic attacks [39]. Designers add rounds until these attacks are infeasible. The third is efficiency, which is traditionally counted in hardware gates or processor cycles. AES scores well on all three and is fast in software because modern processors carry dedicated instructions for it.

None of these goals consider the cost of evaluating the cipher under FHE, since it was not a concern when these ciphers were designed. This leads to likely mismatches between ciphers and homomorphic schemes. A cipher designed for processor throughput can be among the worst possible choices for transciphering, because the operations a processor does cheaply, table lookups and byte permutations, are exactly the ones an FHE backend finds expensive (Section 2.2). This mismatch has led designers to develop new symmetric ciphers specifically optimised for homomorphic evaluation. These ciphers are called *FHE-friendly* ciphers.

2.3.4. FHE-Friendly Design

Under FHE the optimisation goals are different. Section 2.2 showed that cost is set by the number of bootstrapped gates on the TFHE backends, or by multiplicative depth on BFV, not by processor cycles. A cipher built for FHE therefore minimises nonlinear operations and multiplicative depth and can freely incur higher linear costs, which are computationally cheap for a homomorphic backend.

A distinct landscape of FHE-friendly ciphers has emerged around this aim. LowMC was the first block cipher designed explicitly for the multiparty and homomorphic settings, deliberately keeping its AND-count low [20]. The stream cipher Kreyvium [18] keeps only three AND products per keystream bit. The hardware-oriented Grain-128a [34], included here as a second 128-bit stream cipher, carries a larger but still modest nonlinear footprint (Table 5.2). Rasta pushes the idea further, minimising both the number and the depth of its nonlinear operations [35]. The broader landscape is surveyed in the hybrid homomorphic encryption literature [22] and systematised, with benchmarks, in the SoK of Niu et al. [24].

Exactly which of these forms fits which backend, why, and how it is optimised is the question Part I builds the pipelines to answer and the comparison settles. The five ciphers this thesis transciphers are introduced together in Chapter 5, where Table 5.1 summarises each one's structure and best-fit backend, and the chapter treats its homomorphic cost in detail.

2.4. Transciphering

Transciphering combines the two primitives of the previous sections: data travels under a cheap symmetric cipher and computation happens under FHE. This is the idea this thesis uses against the ciphertext expansion of Section 2.1. This section explains the construction and locates the cost it introduces, which Part II then measures.

2.4.1. The Construction

Consider a party holding plaintext data m, for instance the bits of its Bloom filter, and a symmetric key k. Instead of encrypting m under FHE, which would inflate it by the orders of magnitude of Section 2.1, the party uploads two things. The first is its data encrypted under the symmetric cipher, $\mathrm{Sym}_k(m)$, whose

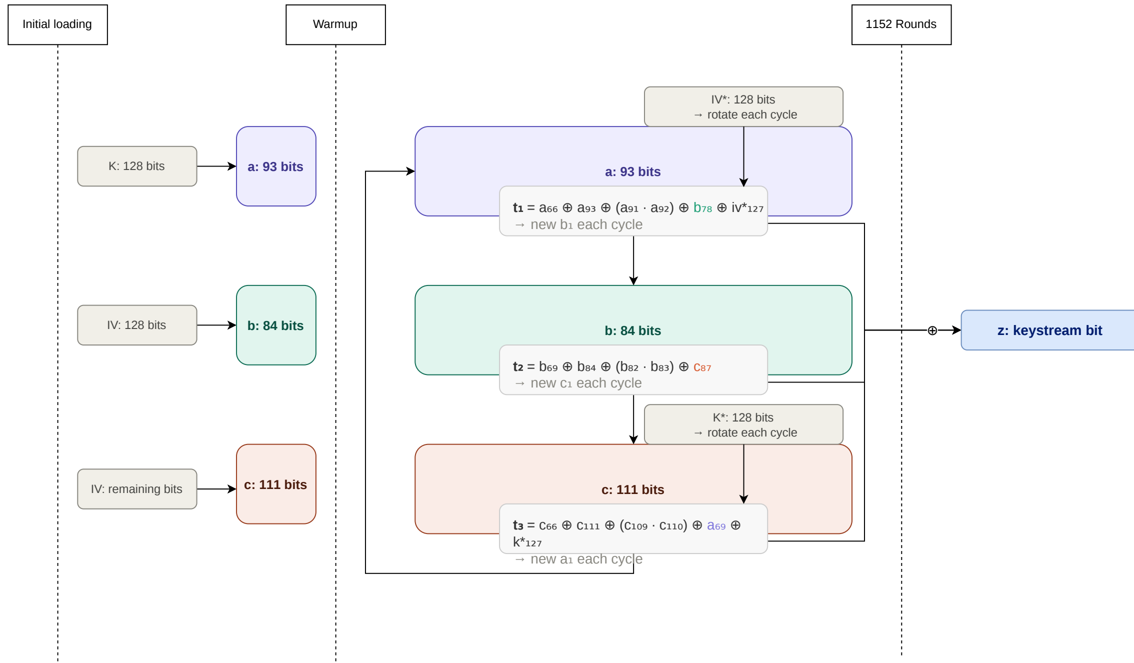


Figure 2.3: In Kreyvium, the homomorphic cost is decided by the operations in its keystream generation: a handful of XORs and 3 ANDs per output bit.

ciphertext is essentially the same size as m. The second is its symmetric key encrypted once under FHE, $\mathrm{FHE}(k)$, a single fixed-size object. Figure 1.2 shows this flow. The pairing of a symmetric cipher for transport with an FHE scheme for computation is known as *hybrid homomorphic encryption* (HHE). It was first proposed to cut the upload to the cloud [19] and later made practical with FHE-friendly ciphers [18].

The effect on the upload is the whole point. A direct-FHE upload costs the size of the data multiplied by the ciphertext expansion factor. A transciphered upload costs the size of the data plus one fixed-size encrypted key, because only the key is sent under FHE. For any data larger than a single key, the second is far smaller.

2.4.2. Homomorphic Evaluation of the Decryption Circuit

The upload is now compact, but the data reaches the server as symmetric ciphertext. This ciphertext is not homomorphic, so the server cannot yet compute on it. Bridging that gap is the *transciphering* step. Holding $\mathrm{FHE}(k)$ and the public symmetric ciphertext $\mathrm{Sym}_k(m)$, the server homomorphically evaluates the cipher's decryption circuit, with the encrypted key as its input and the symmetric ciphertext as public constants. The output is an FHE encryption of m, exactly as if the party had encrypted m under FHE in the first place, since we essentially decrypt the symmetric ciphertext, only homomorphically. From there the server runs the intended computation, in our case the threshold circuit of Chapter 4, just as in the direct-FHE pipeline. At no point has the server seen a plaintext.

This is where the cost moves. Transciphering does not make work disappear. Instead, it trades upload bandwidth for server computation. On top of the intended computation, the server must now homomorphically evaluate the cipher's decryption circuit and, as discussed in Sections 2.2 and 2.3, the price of that circuit is set by the cipher's operation count and multiplicative depth on the chosen backend. A poorly matched cipher makes this expensive. AES, whose S-box is built from the operations FHE finds costly, is slow to evaluate homomorphically, whereas an FHE-friendly cipher keeps the added cost small. The size of this added per-bit cost and how it trades against the bandwidth saved is what Part II measures across the pipelines.

3
Related Work

This thesis sits at the intersection of three fields of research, namely, transciphering and hybrid homomorphic encryption, the design of FHE-friendly symmetric ciphers, and private set intersection. The chapter reviews each in turn, places the work within the wider landscape of privacy-preserving computation, and states what sets this thesis apart. The main observation is that prior work mostly studies the parts of a transciphering pipeline in isolation and on generic circuits, whereas this thesis compares whole pipelines on a real, multi-party application use case, covering both SIMD-friendly and bit-oriented FHE schemes.

3.1. Transciphering and Hybrid Homomorphic Encryption

The idea of transciphering was first proposed by Naehrig, Lauter, and Vaikuntanathan as a way to cut the data a client must upload to a homomorphic cloud [19]. It was made practical by Canteaut et al., who paired a lightweight stream cipher with FHE and framed the technique as a form of homomorphic-ciphertext compression [18]. The resulting construction, a symmetric cipher for transport and an FHE scheme for computation, is now studied under the name hybrid homomorphic encryption, surveyed by Abdinasibfar et al. [22].

Most subsequent work pairs one cipher with one FHE scheme. For the approximate-number scheme CKKS, the Real-to-Finite-field framework introduced the HERA cipher [40] and was followed by the noisy cipher Rubato [41], both aimed at privacy-preserving machine learning over real values. For the word-based schemes BGV and BFV, Masta adapts the Rasta design to modular arithmetic [42] and Pasta makes the case for hybrid homomorphic encryption over a prime field [43]. On the bit-oriented TFHE side, Trivium-on-TFHE shows how a leveled treatment of XOR lowers the per-bit cost [31]. Beyond cipher design, hybrid homomorphic encryption has been applied end-to-end to individual workloads, such as classifying sensitive ECG data on resource-constrained devices [44] and private decision-tree evaluation [45].

What these studies share is their unit of evaluation. Each optimises a single cipher against a single backend, or demonstrates a single application, and reports its results under its own parameters and security level. The single-pairing studies are therefore hard to place side by side. A cipher that looks fast in a CKKS framework and one that looks fast on TFHE are measured against different schemes, different circuits, and different security targets, so the results do not transfer. The recent SoK of Niu et al. closes much of this gap, running over twenty ciphers and several transciphering methods in one environment, and even end-to-end on two real applications [24]. Those end-to-end benchmarks, however, are arithmetic, SIMD-friendly workloads: a genome-wide chi-square test and convolutional-network inference, run on the CKKS and BFV schemes. The study deliberately leaves small and bit-valued workloads, and with them the bit-oriented TFHE backends, out of that comparison. Additionally, both of its applications are single-party. The bit-valued, multi-party workload this thesis targets, threshold PSI evaluated end-to-end across the TFHE Boolean, TFHE shortint, and BFV pipelines, is the case it does not cover, and so it is the gap this thesis addresses.

3.2. FHE-Friendly Symmetric Cipher Design

Underlying the constructions above is a distinct field of symmetric-cipher design, aimed at primitives that are cheap to evaluate homomorphically rather than fast on a processor. LowMC was the first block cipher built explicitly for the multiparty and homomorphic settings, deliberately minimising its AND-count [20]. The stream cipher Kreyvium [18] keeps only a few nonlinear products per keystream bit, and the hardware-oriented Grain-128a [34] remains comparatively light. The algebraic ciphers Rasta [35] and its fixed-layer variant Dasta [46] minimise both the number and the depth of nonlinear operations, and the prime-field designs MiMC [21] and Pasta [43] target the word-based schemes. A more recent strand designs ciphers native to TFHE that operate on small integers rather than bits. They include Elisabeth [47], the small-integer successor of the bit-valued filter-permutator FLIP [48], and the stream ciphers Transistor [49] and FRAST [50].

This thesis relies on this field of study rather than extending it. It does not propose a new cipher, but it takes representative members of these families and asks which one a given FHE backend can transcipher most cheaply for a concrete workload. The TFHE-native small-integer designs are noted but left out of scope, because the workload here is bit-valued and they do not output single bits. Chapter 5 gives the rationale for the cipher selection in full.

3.3. Private Set Intersection

Private set intersection lets parties learn the elements their sets have in common and nothing else. The two-party problem has a long history, from the early homomorphic and polynomial-based construction of Freedman, Nissim, and Pinkas [51] to fast protocols built on oblivious transfer. The multiparty and threshold variants, where an item counts when it appears in at least a chosen number of the parties' sets, were studied by Kissner and Song as privacy-preserving set operations [52]. This thesis uses this threshold PSI idea as a workload.

A directly relevant variant applies homomorphic encryption to PSI in order to reduce communication. Chen, Laine, and Rindal evaluate the intersection under FHE so that the upload is sublinear in the larger set [53], and the labelled and unbalanced extensions, realised in the APSI library, reduce both computation and communication further [54]. These works attack PSI's communication at the level of the protocol, through batching and encodings, rather than by reducing the cost of FHE itself. They are two-party protocols and none combines transciphering with PSI. They are therefore a different approach to the same problem, so a useful alternative for comparison.

The way a set is encoded for homomorphic evaluation also matters. Representing each set as a Bloom filter, a fixed-length, probabilistic bit array, turns the intersection into a bit-wise circuit whose size depends on the filter rather than on the number of distinct items. This comes at the cost of a controlled false-positive rate. Encrypting those Bloom filters and evaluating a threshold sum-of-products circuit over them under FHE is the construction of the Circuit-TMPSI protocol of Vasanthakumari [55], which develops the threshold-MPSI approach to collaborative threat intelligence introduced by Guan [56]. This is the construction that this thesis uses as its threshold-PSI workload and direct-FHE baseline, with one difference in the trust model: where Circuit-TMPSI shares the decryption key across all parties through threshold BFV, this thesis evaluates the same circuit under single-key FHE (Section 4.2). The protocol, its implementation and design choices are described in full in Chapter 4.

3.4. Privacy-Preserving Outsourced Computation

FHE is one of several ways to compute on data without exposing it, and it is worth marking where transciphering sits among them. Secure multiparty computation distributes a computation among several parties so that no single one learns the inputs [57], but it requires the parties to stay online and interact, and its cost is driven by rounds of communication between them. Differential privacy takes a different aim, bounding what any released result can reveal about an individual by introducing noise [58], which protects the output rather than the computation and gives up exact accuracy in return. FHE occupies a distinct point in this space. A single untrusted server computes on encrypted data, non-interactively and with exact results. We adopt this client-server setting because it fits our target workload, and is exactly the deployment model transciphering is designed to make affordable.

Against all of the work above, the contribution of this thesis is one of integration and evaluation rather than a new component. The ciphers and the backends already exist. The gap is the comparison for a bit-valued, multi-party workload across both the SIMD-friendly and bit-oriented backends, together with the framework to carry it out. Building that framework and providing that comparison for threshold PSI is the subject of the remainder of this thesis.

Part I

Building Transieve

4

Threshold PSI and the Baseline FHE Pipeline

Part I builds the pipelines that Part II compares, so this chapter explains the groundwork it is built on: the workload they all evaluate and the direct-FHE baseline they are all measured against. The workload is threshold private set intersection, the collaborative threat intelligence problem of Chapter 1. Its implementation under homomorphic encryption, an encrypted Bloom filter per party with a threshold circuit evaluated over the bits, is an existing construction that this thesis adopts (Section 3.3) [55]. This chapter covers TPSI facets relevant to the thesis: the security model the workload runs under and the baseline implementation of it across the three backends.

Section 4.2 sets out the system and threat model. The two sections that follow give the two halves of the workload's implementation: the Bloom-filter encoding that turns each party's set into individually encryptable bits, and the threshold sum-of-products circuit evaluated over them. The chapter then builds the direct-FHE pipeline, the baseline that encrypts every bit and evaluates the circuit with no symmetric cipher in between. It closes with its theoretical cost in the complexity analysis. That cost is the reference expansion and computation overhead that the transciphering pipelines of Chapter 5 aim to improve on.

4.1. The Threshold-PSI Workload

Threshold private set intersection generalises ordinary set intersection. There are N parties, each holding a private set, together with a threshold T in the range $1 \leq T \leq N$. The result is the set of items that appear in at least T of the N parties' sets, and nothing more. The two extremes are special cases of TPSI. At $T = 1$ the result is the union of all the sets, and at $T = N$ it is the ordinary intersection, where an item must appear in every set. A threshold set between those extremes returns the items held in common by enough parties to be of interest, without demanding all of them. Beyond N and T, the workload is shaped by the size of each party's set, the overlap between the sets, and the Bloom filter's false-positive rate. These are parameters to explore, so Chapter 6 scales them.

In the collaborative threat intelligence setting of Chapter 1, each party is an organisation and its set is the suspicious IP addresses observed in its own logs. An address that surfaces in at least T organisations' sets is evidence of activity wider than a single network, while an address seen by fewer than T stays private. The threshold therefore sets the level of cross-organisational consensus an indicator needs before it is reported.

Threshold PSI is the workload used throughout this thesis, but the pipelines built in Chapter 5 are not specific to it. Every backend evaluates the same object: a slot-wise Boolean sum-of-products over N equal-length bit-vectors, one per party, producing one result bit per position. Any workload whose computation can be cast in this form, a per-position Boolean function of N bit-vectors, can run on the same pipelines without change. That enables a family of set- and vote-style protocols to be

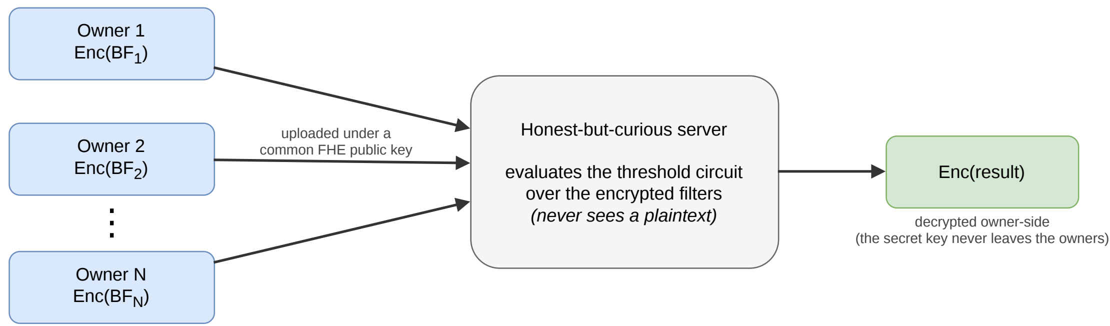


Figure 4.1: The cloud-assisted system model: owners upload encrypted filters, the server computes the threshold result without ever decrypting.

easily implemented in Transieve, and Section 5.5 demonstrates it with a second workload that uses a different encoding and no Bloom filter. Workloads that need integer arithmetic, such as exact counts or magnitude comparisons, fall outside this Boolean model and are out of scope.

4.2. System and Threat Model

Gaining the bandwidth advantage from transciphering must not impact the workload's security model, so this section defines the model the rest of the thesis assumes. It is deliberately simpler than the original Circuit-TMPSI construction [55]. Where Circuit-TMPSI splits the decryption key across all parties, this thesis uses a single FHE key and focuses on confidentiality against the compute server. The impact of this decision is described below.

4.2.1. Parties and Architecture

The setting has N data owners and a single compute server. Each owner is an organisation holding a private set and the server is a cloud provider that runs the homomorphic computation. An owner communicates only with the server, never with the other owners. Each owner uploads its data encrypted under a common FHE public key, the server evaluates the threshold computation over the ciphertexts, and the encrypted result is returned to be decrypted into the output. The scheme is single-key, so one FHE key pair is used throughout. The secret key is held on the owner side, which performs the final decryption. The server never holds it. Figure 4.1 shows this flow.

4.2.2. Adversary Model

The server is assumed honest-but-curious: it follows the protocol faithfully but may inspect anything that passes through it. This is the standard assumption for cloud or outsourced computation, where a provider runs the requested job correctly yet should not be trusted with the plaintext. The server observes the FHE ciphertexts, the public parameters, and the circuit it evaluates. Additionally, in the transciphering pipelines, the server sees each party's symmetric ciphertext and FHE-encrypted symmetric key. In no case does it see a plaintext set or result.

4.2.3. Security Goals

The goal is input confidentiality against the server: no owner's set is revealed beyond what the agreed output already discloses. That output is the threshold result alone, the items meeting the threshold, and says nothing further about which owner holds which item or what else any set contains. The Bloom-filter encoding adds one caveat, however, which is that the recovered result may contain false positives. This is a matter of accuracy rather than confidentiality and is quantified in Chapter 6.

4.2.4. Assumptions and Scope

Confidentiality in this setting rests on the hardness of (Ring) Learning With Errors, with all parameters chosen for $\lambda = 128$-bit security (Section 2.2). Three decisions here impact the model. First, the scheme is single-key rather than threshold decryption. This results in a weaker trust guarantee than a threshold scheme in which no single party can decrypt. The choice does not affect what the thesis

measures, however. Transciphering is the same homomorphic computation on the same-sized ciphertexts whichever key model is used, and threshold FHE differs only in key generation and decryption: one-time stages that are identical across the compared pipelines. The server-side evaluation cost and the bandwidth results therefore largely transfer to a threshold deployment, so single-key is a simplification rather than a constraint on the comparison. Second, the symmetric ciphers are taken at their claimed security. Their own cryptanalysis is out of scope. These designs attract ongoing cryptanalysis, for example [59], which we do not factor in. Third, a malicious server, side channels, and traffic analysis are out of scope. These last boundaries are the usual ones for measuring transciphering cost. Strengthening the adversarial model is left to future work.

4.3. Bloom-filter Encoding

A Bloom filter represents a set as a fixed-length array of M bits, initially all zero, addressed by K hash functions [60]. Inserting an element sets the K bits its hashes select, and testing membership checks those same bits. The structure never reports a false negative but there is a controlled rate of false positives. Its length M and hash count K follow from the set size and the target false-positive rate by the standard formulas:

$$M = \left\lceil \frac{-S \ln \varepsilon}{(\ln 2)^2} \right\rceil, \qquad K = \operatorname{round}\left(\frac{M}{S} \ln 2 \right)$$

where ε is the per-filter false-positive target.

A variable-size set becomes a fixed M-bit array, so the threshold circuit has the same size and shape regardless of how many items a party holds. Its cost depends only on M. This makes the protocol a good fit as a workload.

What the encoding must provide here is individual access to every bit. The threshold circuit of the next section is evaluated position by position, so each filter bit has to be separately encryptable: a mapping of one FHE ciphertext per bit on the TFHE backends, or one slot of a packed ciphertext on BFV. General-purpose Bloom-filter libraries expose only insertion and membership queries, not the underlying bit array, so this thesis uses a custom filter that lays the bits out explicitly and exposes them for encryption.

The same hash functions are shared by all parties, and the filter is fully deterministic. Bit positions come from a keyed hash (SipHash) with fixed keys, combined by double hashing to give the K indices. A given item therefore maps to the same positions in every party's filter, which is what makes the per-position threshold meaningful. An item held by several parties sets the same bits in each of their filters, so the circuit can apply the threshold one position at a time. Determinism also lets the plaintext evaluation serve as the correctness oracle for the homomorphic path, which must reproduce it bit for bit.

4.4. The Threshold SOP Circuit

The computation at the heart of the workload is a threshold test applied to every Bloom-filter position. Party i holds a filter BF_i, and the result filter BF^* is defined position by position: $\mathrm{BF}^*[j] = 1$ exactly when at least T of the N parties have a one at position j. An item is then reported when all of its K positions are set in BF^*, recovered by testing it for membership against BF^*. A true threshold item, present in at least T parties, sets each of its positions in each of those parties' filters, so every one of its positions clears the threshold and the item is always recovered. The Bloom structure can additionally admit false positives.

At each Bloom-filter position the inputs are the N parties' bits at that position, $\mathrm{BF}_1[j], \ldots, \mathrm{BF}_N[j]$, and the per-position test is the Boolean threshold function: it returns one when at least T of them are one. Written straight from its truth table, as one term per satisfying assignment, the function is exponentially large and full of negations. Vasanthakumari shows that it collapses, through repeated application of

the idempotent and absorption laws, to a minimal sum-of-products with no negations [55]:

$$C_T\big(\mathrm{BF}_1[j], \dots, \mathrm{BF}_N[j]\big) \;=\; \bigvee_{\substack{A \subseteq \{1,\dots,N\} \\ |A|=T}} \; \bigwedge_{i \in A} \mathrm{BF}_i[j].$$

This is the circuit evaluated at every position of the result filter, and it is simply a sum-of-products. Each product is the AND of the T parties in one size-T subset, there is one product for each of the $\binom{N}{T}$ subsets, and the sum is the OR over all of them. For $N = 3$ and $T = 2$ the three subsets $\{1,2\}, \{1,3\}, \{2,3\}$ give

$$\mathrm{BF}^*[j] \;=\; \big(\mathrm{BF}_1[j] \wedge \mathrm{BF}_2[j]\big) \vee \big(\mathrm{BF}_1[j] \wedge \mathrm{BF}_3[j]\big) \vee \big(\mathrm{BF}_2[j] \wedge \mathrm{BF}_3[j]\big).$$

This thesis adopts this circuit unchanged.

From the formula, the circuit holds $\binom{N}{T}$ AND-terms, each of T literals, and that count is what governs its homomorphic cost. It is smallest when T is near 1 or N and largest when T is near $N/2$, where $\binom{N}{T}$ peaks. Chapter 6 traces this directly in the scaling behaviour.

Each backend implements this Boolean circuit with its own primitives. BFV uses arithmetic over the plaintext field, where an AND is a product and an OR is $a \vee b = 1 - (1-a)(1-b)$, and the TFHE backends use bootstrapped gates, as explained in Chapter 2. The same circuit is applied identically to every Bloom-filter position, which is what lets the bit-parallel TFHE backends and the slot-parallel BFV backend evaluate it without per-position branching.

4.5. The Direct-FHE Pipeline

The direct-FHE implementation evaluates the workload under FHE with no symmetric cipher involved, as a baseline. Each party encrypts its Bloom filter bit by bit under the common public key and uploads the resulting ciphertexts. The server evaluates the threshold sum-of-products over the encrypted filters, one position at a time, and returns the encrypted result filter. The result filter is decrypted on the owner side, and the reported items are read off by testing membership against it. The plaintext evaluation of the same circuit runs in parallel as the correctness oracle, which the decrypted result must match bit for bit.

All transciphering pipelines run the direct-FHE as well. Each of the three backends is run with the same circuit, varying only the primitives. On the TFHE backends each Bloom bit is its own ciphertext, and the AND and OR gates are bootstrapped. On BFV the bits are packed into the slots of a ciphertext and the circuit is evaluated as slot-wise field arithmetic. Because every position runs the identical circuit, the per-position cost does not depend on the filter length, which Chapter 6 uses to measure cost on a sampled prefix and extrapolate to the full filter.

This direct path is the reference the rest of the thesis is measured against. Its server work is the threshold circuit alone, with no cipher to undo first, and its upload is the entire filter sent as FHE ciphertext, carrying the full expansion of Section 2.1.

4.6. Complexity Analysis

The baseline's cost is the reference that Chapter 5 measures the transciphering pipelines against. Two quantities define it, namely how much each party uploads and how much the server computes. Both are given here in terms of the number of parties N, the threshold T, and the Bloom-filter length M.

4.6.1. Communication

Each party encrypts its M-bit Bloom filter and uploads it, so the upload is M ciphertexts per party and $O(MN)$ in total. The server returns the M-bit result filter for decryption. The count is not the real cost, however, the size of each ciphertext is. Every plaintext bit travels as a full FHE ciphertext, so the upload carries the ciphertext expansion factor F of Section 2.1, around an order of magnitude for densely packed ciphertexts and several orders of magnitude when each bit is sent separately. This expansion is what transciphering attempts to mitigate, and it is the baseline against which the upload saving of Chapter 5 is reported.

4.6.2. Computation

The server evaluates the threshold sum-of-products at each of the M positions. A single position is an OR of $\binom{N}{T}$ products of T bits, so it costs $O\big(T\binom{N}{T}\big)$ homomorphic operations and the whole filter $O\big(M\,T\binom{N}{T}\big)$. The $\binom{N}{T}$ factor dominates and peaks at $T \approx N/2$. How that operation count becomes wall-clock time depends on the backend, as Chapter 2 explained. On the TFHE backends every AND and OR is a bootstrap, so the cost tracks the total gate count. On BFV the cost is set not by the gate count but by the circuit's multiplicative depth, the longest chain of ciphertext multiplications it requires. An AND is one multiplication and an OR, written $1-(1-a)(1-b)$, contains one, while additions are almost free. That chain has two stages. Forming a single T-way AND term as a balanced tree of products takes $\lceil \log_2 T \rceil$ multiplications, and OR-ing the resulting $\binom{N}{T}$ terms together, again as a balanced tree, adds a further $\lceil \log_2 \binom{N}{T} \rceil$, for a total depth of $\lceil \log_2 T \rceil + \lceil \log_2 \binom{N}{T} \rceil$. Each extra level of depth forces a larger ciphertext modulus to absorb the accumulated noise, and it is this growth that eventually meets the BFV security ceiling examined in Chapter 6. The per-position circuit is identical at every position, so the total is linear in M and the per-bit figure is well defined.

Together the baseline carries a large upload and a server cost equal to the threshold circuit alone. Transciphering keeps that circuit but evaluates a cipher's decryption ahead of it, and replaces the direct-FHE upload of MN encrypted Bloom-filter bits, or $O(NM/slots)$ packed ciphertexts on BFV, with a plaintext-sized symmetric ciphertext and one encrypted key. Chapter 5 analyses that trade against the reference set here.

5
Transciphering Pipelines

Chapter 4 built the direct-FHE baseline. This chapter builds the transciphering pipelines that set out to improve on it, with respect to ciphertext expansion. Each keeps the threshold computation of the baseline but changes how the data reaches the server. Instead of uploading the data as FHE ciphertext, a party sends it under a lightweight symmetric cipher, together with its key encrypted once under FHE. Then the server recovers the FHE ciphertexts by homomorphically evaluating the cipher's decryption. The upload shrinks to roughly the plaintext size, and the server pays for it in the added cost of that decryption.

There is one pipeline per backend, and all three are built on a shared framework. Section 5.1 describes that framework and how a pipeline is assembled from a workload, a cipher, and a backend. The three sections that follow build the Boolean, shortint, and BFV pipelines, each paired with a cipher that suits it. Section 5.5 then shows the same framework carrying a second workload, and the chapter closes with the per-pipeline complexity, set against the baseline of Chapter 4.

5.1. Designing Pipelines

A deployable transciphering system is not a cipher or a backend on its own, but the whole pipeline: an FHE backend, the symmetric cipher's decryption circuit, and the evaluation of the workload's circuit on the recovered ciphertexts. As Chapter 2 argued, these choices form a chain. The workload fixes the shape of the computation, the backend that carries it efficiently follows, and the backend in turn decides which ciphers transcipher into it cheaply. Designing a pipeline therefore means working down this chain, not choosing parts in isolation.

5.1.1. The Transieve framework

To keep those pipelines comparable rather than three separate prototypes, they are built within a single framework, *Transieve*. Transieve is organised as three layers with narrow interfaces between them, shown in Figure 5.1:

- a **workload layer** that holds the application: how its data is encoded (for threshold PSI it is the Bloom-filter encoding of Chapter 4) and the circuit to be evaluated under FHE (the threshold sum-of-products);
- a **transciphering layer** that holds the symmetric ciphers and their homomorphic decryption circuits, behind a common interface so that every cipher exposes the same operation: obtain FHE ciphertexts from a symmetric ciphertext and an FHE-encrypted key by homomorphically decrypting;
- an **FHE backend layer** that holds the concrete schemes, TFHE Boolean, TFHE shortint, or BFV, and the primitive operations the layers above are written against.

The concrete interfaces each layer exposes are listed in Appendix B.

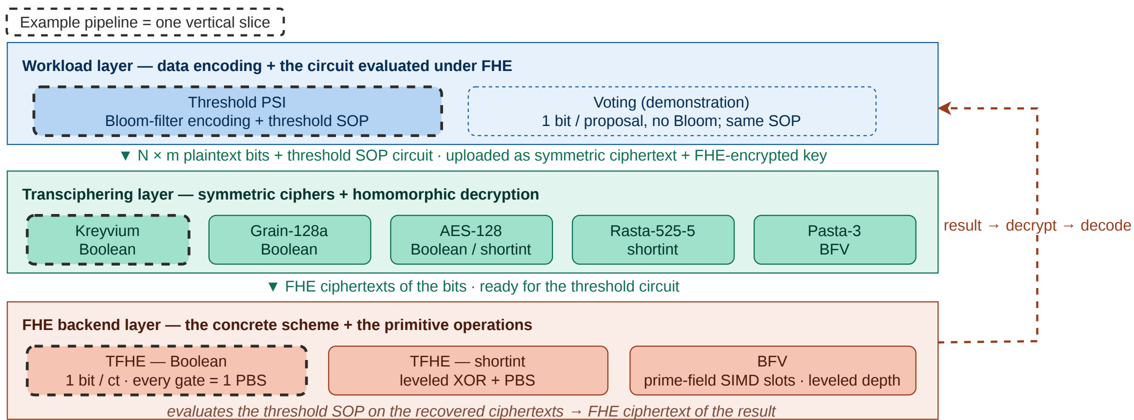


Figure 5.1: The Transieve framework. Three decoupled layers: workload, transciphering, homomorphic backend

The separation turns the three choices in the chain into three independent modules. A pipeline is one vertical slice through the layers: a workload on top, a chosen cipher in the middle, a chosen backend underneath. Each pipeline inherently holds a direct-FHE baseline, which is obtained by disabling the transciphering layer. The comparison then varies a single choice while holding the rest fixed, under the identical metrics of Section 6.2, which is what makes it fair.

5.1.2. Extending the Framework

Because the layers meet only at their interfaces, Transieve can be extended along any one of them without friction with the others. A new symmetric cipher is added by implementing its homomorphic decryption against the transciphering interface. It then runs on the backend it targets and against every workload already present. A new FHE backend is added by providing the primitive operations the connecting layers expect. A new workload is added by supplying its encoding and circuit, and immediately inherits every cipher and backend. Section 5.5 demonstrates this with a second workload. This is the property that lets Transieve serve as a reusable evaluation tool beyond the pipelines compared here, not only as an evaluation framework for this thesis.

5.1.3. Cipher Selection

The chain fixes the order of selection: the workload chooses the backend, and the backend chooses the cipher. Section 2.2 shows which algebra each backend makes cheap. This section outlines the ciphers chosen for each backend and those ciphers left out of scope.

Five ciphers are transciphered, each matched to where it fits. AES-128 is the standardised baseline and chosen for that reason. It is a symmetric cipher in the SPN family. Its S-box is finite-field inversion and its rounds are dense in the table lookups and bit permutations FHE finds expensive, so it anchors the comparison from the costly end [32]. The low-gate stream ciphers Kreyvium [18] and Grain-128a [34] spend only a few nonlinear products per keystream bit and suit the bit-oriented Boolean backend. Rasta [35] minimises nonlinear count and depth but pays for it in dense linear layers, which makes it a poor fit for Boolean but a natural one for shortint, where those layers become cheap. Pasta [43] is a prime-field cipher, the GF(p) descendant of the Rasta line, and the match for BFV. A GF(2) cipher like Rasta cannot be SIMD-batched on the prime-field backend, so it would forfeit the slot parallelism that BFV enables.

Other symmetric ciphers were also considered. LowMC [20] optimises multiplicative depth for leveled schemes rather than the total gate count the TFHE backends charge for, and its security analysis is less settled than Rasta's, which fills the same niche in its nonlinear layer. Keccak used as a cipher adds no nonlinearity beyond the χ layer Rasta already uses, at higher cost. The TFHE-native small-integer ciphers, Transistor, FRAST, and Elisabeth, do not output single bits and so mismatch the bit-valued workload (Section 3.2).

The outcome is a set of matched pairings rather than every cipher on every backend, and the gaps

Table 5.1: Overview of the ciphers compared.

Cipher	Family	Nonlinear layer	Linear layer	Best-fit backend
AES-128	Block, SPN	GF(2^8) inversion S-box	ShiftRows / Mix-Columns (GF(2))	Boolean (anchor)
Kreyvium	Stream, NLFSR	few AND taps per bit	sparse shifts + XOR	Boolean
Grain-128a	Stream, NLFSR	AND taps per bit	sparse shifts + XOR	Boolean
Rasta-525-5	Stream, algebraic	Keccak χ	dense random affine (GF(2))	shortint
Pasta-3	Stream, algebraic	cube + Feistel S-box	matrix–vector (GF(p))	BFV

carry information of their own. A prime-field cipher such as Pasta would have to be bit-decomposed to run on the bit-oriented backends, discarding its advantage, while a GF(2) cipher cannot be slot-batched on the prime-field BFV library and AES's GF(2^8) S-box would need byte-level batching the library used does not provide. These conversions are out of scope. Each pipeline therefore carries a cipher its backend's algebra actually fits, and Chapter 6 measures how much that match is worth.

Table 5.1 summarises the five ciphers by structure and the backend each is matched to. Two attributes decide homomorphic cost: the nonlinear layer fixes the bootstrap count on the TFHE backends and the multiplicative depth on BFV, while the linear layer is what makes an XOR-dense design such as Rasta expensive on Boolean (every XOR is a bootstrap) yet affordable on shortint (where XOR is leveled). The per-pipeline cost these imply is quantified in Table 5.2.

The resulting comparison consists of six transciphering pipelines. They are not meant to form a complete catalogue of all possible cipher/backend pairs. Instead, they cover the main scenarios needed for this thesis: bit-oriented stream ciphers on Boolean TFHE, standardised AES anchors on both TFHE backends, an algebraic cipher whose structure fits the shortint backend, and a prime-field cipher designed for SIMD evaluation under BFV. The evaluated pipelines are:

- **Kreyvium/Boolean**: a compact-upload Boolean pipeline using an FHE-friendly stream cipher with a low per-bit nonlinear cost.
- **Grain-128a/Boolean**: a second stream-cipher Boolean pipeline, included to compare Kreyvium against another 128-bit FHE-friendly stream cipher.
- **AES-128/Boolean**: a standardised block-cipher anchor on the bit-oriented backend, included to show the cost of using a conventional cipher under Boolean FHE.
- **AES-128/shortint**: the same standardised cipher on the shortint backend, used to test whether grouping operations into small integers improves the AES transciphering path.
- **Rasta-525-5/shortint**: an algebraic-cipher pipeline included as a negative result; shortint avoids the Boolean cost of dense XOR layers, but the resulting pipeline remains too expensive to be competitive.
- **Pasta-3/BFV**: a prime-field SIMD pipeline whose cipher and backend share the same arithmetic structure, giving the lowest measured per-bit transciphering cost but a larger upload and a leveled-depth ceiling.

5.2. The Boolean Pipeline

The Boolean backend holds one bit per ciphertext, and every binary gate, AND, OR, and XOR alike, costs one bootstrap (Section 2.2). Only negation is free, since it is a linear operation on the ciphertext that does not increase the noise. The transciphering cost on this backend is therefore the cipher's total gate count per output bit, not its nonlinear count alone. For a stream cipher that cost sits entirely in keystream generation. The server clocks the cipher's shift registers homomorphically to produce keystream bits, then XORs them with the public symmetric ciphertext, which is free because that ciphertext is a plaintext constant. A block cipher in counter mode is handled the same way, with its round function producing the keystream. Keystream uniqueness is provided by the key rather than the nonce:

each party transciphers under its own symmetric key, so keystreams never collide across parties, and within a party the whole Bloom filter is recovered from a single non-repeating keystream. The measured implementation fixes the public initialisation value to the all-zero test vector.

The four ciphers on this backend span the cost range. Kreyvium and Grain-128a clock narrow shift registers with a handful of taps and a light nonlinear function per bit, so their gate count per bit is small and the backend handles them comfortably. AES-128 is far heavier: each round applies the finite-field-inversion S-box and then the dense GF(2) diffusion of ShiftRows, MixColumns, and AddRoundKey, so a single output bit pulls in the S-box gates plus a long tail of bootstrapped XORs from the linear layers. The S-box itself is the size-optimised Boyar–Peralta combinational circuit [61], 115 gates of which 32 are AND, so the measured AES cost reflects that optimised netlist rather than a naïve table lookup. Rasta is the other extreme on the Boolean TFHE backend. Because it is defined over GF(2) and the workload is bit-valued, by algebra it looks like the natural Boolean cipher. That apparent fit is what makes its result revealing in Section 7.1. Its design wraps thin nonlinear layers in wide random affine layers, which keeps its nonlinear count and depth low but turns nearly every gate into a bootstrapped XOR. On the Boolean backend Rasta is therefore a negative result. It is correct but far too slow to be practical. The next section shows where the shortint backend tries to rescue it.

Each pipeline runs the full path, transciphering followed by the threshold circuit, but the cipher's contribution is also measured on its own through a microbenchmark that times keystream recovery in isolation. Separating the two keeps the per-bit transciphering cost, the quantity the comparison rests on, free of the workload's own circuit cost. Chapter 6 reports both costs.

5.3. The Shortint Pipeline

The shortint backend holds a small integer per ciphertext, split into a message slot and a carry slot, and that changes which operations are cheap (Section 2.2). An XOR of bits becomes an addition that accumulates in the carry slot with no bootstrap and a bootstrap is performed only to clear the carry back to a clean bit or to evaluate an actual nonlinear step [31]. The transciphering cost on this backend is therefore the cipher's nonlinear count plus an occasional carry clean, rather than its total gate count.

The shortint backend exists because of what the Boolean pipeline exposed. Rasta was paired with Boolean on the expectation that a GF(2) cipher and a bit-valued workload would suit the bit-native backend. The measured negative result overturned that, showing a shared domain is necessary but does not predict cost (Section 7.1). What decides cost is how a backend charges for the operations a cipher actually spends its budget on, so this backend was added to give Rasta's XOR-dense design a home that charges for XOR differently. The dense affine layers that make it a negative result on Boolean are XORs, so on shortint they collapse from a bootstrap each to a leveled accumulation, leaving the handful of nonlinear gates to dominate. The shortint backend is thus the worked rescue of the Boolean Rasta result: the same cipher and the same circuit, made viable by counting XOR as leveled. AES is a perfect counter-example to this combination. It is dense in both XOR and nonlinear gates, so moving it to shortint cheapens its linear layers but leaves the S-box cost untouched, confirming that the leveled-XOR carry trick does not rescue a cipher whose cost is nonlinear.

The shortint backend stays secondary due to the nature of the workload. The threshold sum-of-products is pure bit-AND and bit-OR with no long XOR chains to amortise, and a shortint bootstrap is slower than a Boolean one, so evaluating the circuit itself is a net loss on shortint. The backend earns its place only by rescuing the XOR-dense cipher that Boolean cannot afford, which is the comparison's clearest illustration that the right backend depends on the cipher.

5.4. The BFV Pipeline

The BFV pipeline pairs the backend with Pasta, the one prime-field cipher in the comparison, and works throughout over the field $\mathbb{F}_p$ with $p = 65537$. Two features shape it, taken in turn below: it runs leveled, without bootstrapping, and it packs data into SIMD slots. The depth that follows from running leveled then produces a security ceiling unique to this backend.

5.4.1. Leveled BFV

The BFV pipeline differs from the two TFHE pipelines in one design choice. It runs leveled, with no bootstrapping. This means that the depth it has to support must be known before the run. The multiplicative depths of the Pasta decryption circuit followed by the threshold sum-of-products are both fixed once the parameters are chosen. Sizing the modulus chain to that depth lets the pipeline evaluate the whole circuit without needing a noise refresh [12]. A BFV bootstrap is among the most expensive operations in homomorphic encryption. It runs far slower than an ordinary homomorphic multiplication. The bootstrapping circuit is itself deep, consuming much of the modulus budget it runs in [62]. Avoiding it wherever the depth is bounded is both standard practice and the far cheaper path [10]. The library the backend builds on, `fhe.rs` [26], provides this leveled framework. Bootstrapping BFV would mean moving to a different library, like OpenFHE or SEAL, over a C++ interface, a dependency the implementation was built to avoid.

Bootstrapping BFV would lift the depth bound, but at a cost this thesis does not measure, so it stays out of scope. The consequence of running leveled, a hard limit on circuit depth, is taken up in Section 5.4.3. The BFV pipeline is therefore compared as a leveled scheme against the per-gate-bootstrapped TFHE pipelines [15], matching how each is used in practice.

5.4.2. Pasta and SIMD Evaluation

Pasta is implemented for the BFV backend. Its state is a vector of $\mathbb{F}_p$ elements, which map onto the CRT slots of a single BFV ciphertext, so one homomorphic operation acts on the whole state at once. Its nonlinearity is a low-degree power map over the field. The cube S-box is arithmetised depth-aware [63], keeping it cheap in multiplicative depth. Each S-box is a straight-line arithmetic circuit over $\mathbb{F}_p$ applied slot-wise to the packed state. The cube is evaluated as $x^2 = x \cdot x$ (one ciphertext–ciphertext multiplication, one level) followed by $x^3 = x^2 \cdot x$ (a second level), so it has multiplicative depth two. The Feistel S-box $S'(y)_l = y_l + (y_{l-1})^2$ is a single squaring, depth one. The affine layers are products by public plaintext diagonals and consume no ciphertext–ciphertext level. Following the depth-aware view of Vos et al. [63], the limiting cost is therefore the longest dependency chain of ciphertext–ciphertext multiplications, not the total number of field operations. We apply this principle to Pasta's power maps. In an r-round instance this chain is the $r - 1$ Feistel squarings followed by the final cube, and the implementation sizes the modulus chain at 4 + 3r RNS limbs for the cipher, plus one further limb per multiplicative level of the workload circuit evaluated after it. Table 5.2 reports this chain length ($r+1 = 4$ for Pasta-3). Its linear layer is a matrix–vector product evaluated by the diagonal baby-step/giant-step method [64], which cuts the rotation count from t to about $2\sqrt{t}$: for Pasta-3's $t = 128$ words the split is $t_1 = \lceil \sqrt{t} \rceil = 12$ baby steps and $t_2 = 11$ giant steps. The implementation packs both branches of the state into the two rows of the same Halevi–Shoup hypercube [64], so one column rotation acts on both at once, and tiles several independent keystream blocks across the remaining slots so that a single homomorphic permutation produces many keystream words at once. Both packing schemes are Pasta's own SIMD-evaluation techniques [43]. Concretely, a ciphertext of ring degree n is viewed as a $2 \times (n/2)$ grid: the two branches x_L and x_R occupy the two rows, a column rotation shifts both branches together, and the fixed 2×2 MDS mix becomes a single row swap. These optimisations are what bring the measured Pasta/BFV cost into the few-milliseconds-per-bit range reported in Chapter 6.

What is relevant for the pipeline is that Pasta's recovered output is already slot-packed, in the same layout the threshold sum-of-products accepts. Most transciphering produces coefficient-encoded ciphertexts, which need an expensive coefficient-to-slot conversion, itself consuming depth, before a slot-wise application can use them [22]. Pasta is SIMD-native and avoids that step. This is a property of Pasta's design that the BFV pipeline exploits.

5.4.3. The Security Ceiling

Running leveled bounds the multiplicative depth, and on this backend that bound becomes a hard limit on the party count. The plaintext modulus $p = 65537$ allows full SIMD batching only up to ring degree 32768, and at that degree a 128-bit-secure modulus chain holds at most about 881 bits [23]. The threshold circuit's depth grows with $\lceil \log_2 \binom{N}{T} \rceil$,so the chain it requires climbs with the party count. The budget is fixed: at degree 32768 a 128-bit-secure modulus holds about 881 bits [23], and leveled BFV spends roughly one $\sim$55-bit prime per multiplicative level. On top of the levels the chain carries two further primes, one consumed by the noise of the fresh encryption and one serving as the key-

switching modulus, so a depth-d circuit provisions $d + 2$ primes in total. The direct-FHE baseline then stays securable through $N = 12$, the largest party count in the sweep (a depth-13 chain, ≈ 825 bits), with the budget admitting roughly one more party, $N \approx 13$ at ≈ 880 bits, before the chain exceeds what any secure degree can hold. Transciphering stacks Pasta's own depth on top of the circuit's, so the combined chain consumes the budget much sooner, making the Pasta pipeline only securable to about $N = 4$. Beyond the limit the pipeline returns an error rather than dropping below 128-bit security, so the wall is exact, not estimated. This ceiling is a cost of transciphering itself rather than of FHE in general, and is shown in Chapter 6.

5.5. Generalising the Workload

The workload sits behind the same kind of narrow interface as the cipher and the backend, so it can be replaced as well. A workload provides two things: an encoding that turns its input into the per-party bit-vectors the pipelines consume, and the Boolean circuit evaluated over them. Threshold PSI supplies the Bloom-filter encoding of Chapter 4 and the threshold sum-of-products. Any other workload of the same shape, a per-position Boolean function of N bit-vectors, runs on the existing pipelines unchanged.

A secure voting tally demonstrates this. N voters each cast a yes or no on each of several proposals and the tally reports the proposals carried by at least T of them: the same threshold structure with a different meaning. The encoding is direct, one bit per proposal per voter and no Bloom filter, so the decoding reads passed proposals rather than recovered addresses, yet the circuit is the same threshold sum-of-products and the pipelines beneath it are untouched. The voting workload is implemented as a demonstration that the workload layer is swappable, not as a second evaluated case. Threshold PSI remains the workload the comparison measures. This abstraction is what lets Transieve serve as a reusable evaluation framework. Results of the voting demonstration are shown in Appendix D.

5.6. Complexity Analysis

This section gives the transciphering pipelines' cost in the same terms as the baseline of Section 4.6, so the two can be set side by side. Transciphering changes both quantities the baseline established: it shrinks the upload, and it adds to the server's work.

5.6.1. Communication

A transciphered upload is the data under the symmetric cipher, whose ciphertext is essentially the size of the plaintext for the bit- and byte-oriented ciphers (a field-oriented cipher such as Pasta maps each input bit to an $\mathbb{F}_p$ word and so enlarges the symmetric ciphertext, as Section 6.4 quantifies), together with the symmetric key encrypted once under FHE. Per party this is $O(M)$ plaintext-sized bits and one fixed-size encrypted key, against the baseline's M encrypted Bloom-filter bits, represented as M ciphertexts on the bitwise TFHE backends or packed into slots on BFV. The expansion factor therefore falls from the orders of magnitude of the direct path toward one, with the fixed cost of the encrypted key amortising as the data grows. How that saving scales with data size is the headline communication result of Chapter 6.

5.6.2. Computation

The server's work is the baseline threshold circuit with the cipher's homomorphic decryption added ahead of it. That decryption cost is per output bit and depends entirely on the pipeline: on the Boolean backend it is the cipher's total bootstrapped gate count, on shortint its nonlinear count with leveled XOR, and on BFV its multiplicative depth over the packed slots. The transcipher server cost is therefore the baseline cost plus a per-bit term set by the cipher and backend, and the comparison is largely a comparison of that term. On BFV there is a further effect, in that Pasta's decryption adds its depth to the circuit's, which is what lowers the security ceiling of Section 5.4.3. Chapter 6 measures the per-bit decryption cost for each pipeline and sets it against the bandwidth it buys.

5.6.3. Hardware-Independent Cost

The per-bit latencies of Chapter 6 are specific to the measurement hardware. Two quantities are not: the programmable bootstraps (PBS) a pipeline performs per output bit and its multiplicative depth. Both follow from the cipher's specification and the backend's cost model, so they order the pipelines indepen-

dently of any machine. Table 5.2 lists them for the evaluated pipelines, taken from the cipher constants in the implementation.

Table 5.2: Specification-derived, hardware-independent cost of each evaluated pipeline: nonlinear operations per output bit and multiplicative depth, taken from the cipher constants in the implementation (Appendix B)

Cipher	Backend	Nonlinear ops / bit	Multiplicative depth
Kreyvium	Boolean	3	1
Grain-128a	Boolean	20	3
AES-128	Boolean	40	4
AES-128	shortint	40	4
Rasta-525-5	shortint	5	5
Pasta-3	BFV	0[a]	4[b]

[a] Pasta's cube and Feistel S-boxes are nonlinear but leveled, so they cost no bootstraps.
[b] Two depth-1 Feistel squarings followed by the depth-2 cube (Section 5.4.2).

The bootstrap count must be read against how each backend bootstraps. On shortint a bootstrap is performed for each nonlinear step and each carry cleanup, while the XOR-dense linear layers accumulate without bootstrapping. Rasta's eleven bootstraps per bit are its five nonlinear χ operations plus six carry cleanups. The bootstrap count bounds the nonlinear work, but it does not fix the wall-clock cost on its own: the affine rows still cost roughly 1,500 leveled ciphertext additions per output bit, and although each addition is far cheaper than a bootstrap, at that volume the leveled layer remains the dominant share of Rasta's measured shortint time. Chapter 6 makes this visible: the measured ordering of the two shortint pipelines turns out to be the reverse of their bootstrap ordering, and Section 7.1 explains why. For linear-heavy designs the bootstrap count must therefore be read together with the leveled-operation volume. On BFV the binding quantity is multiplicative depth, because the scheme is leveled and the linear layer is plaintext diagonals that add no ciphertext-by-ciphertext depth. Pasta therefore performs no bootstraps and spends two Feistel squarings and one cube (a depth-4 chain, Table 5.2). The Boolean backend is read differently again. Every gate is bootstrapped there, so the physical bootstrap count per bit is the cipher's *total* gate count, and the figures above are its nonlinear (AND) gates alone. For the low-XOR stream ciphers that gap is small, but for the linear-heavy designs it is large: the same Rasta has only five nonlinear operations per bit, yet its roughly 262-ones affine rows expand into on the order of fifteen hundred bootstrapped gates per bit, which is why it is a Boolean negative result and is evaluated on shortint instead. AES's forty nonlinear gates likewise sit inside a far larger total-gate cost on Boolean. The counts therefore order the ciphers and explain the rankings of Chapter 6: they estimate shortint cost through the bootstrap count where the leveled linear volume is small, and BFV cost through the depth, while on Boolean they must be read together with the total-gate condition in mind.

Part II

Evaluating Transciphering on TPSI

6 Experiments and Evaluation

This chapter evaluates the transciphering pipelines of Part I on the threshold private set intersection workload, comparing each complete pipeline under one fixed set of metrics. The comparison is end-to-end: every pipeline is measured as a whole rather than by any single component.

The chapter is organised as follows. Section 6.1 states the security basis under which the pipelines are compared. Section 6.2 describes the experimental setup, datasets, parameter sweeps, and metrics. Section 6.3 reports the plaintext protocol sweeps, which isolate the Bloom-filter and threshold-circuit behaviour from FHE cost. Section 6.4 compares the FHE and transciphering pipelines on server compute and upload bandwidth. Section 6.5 reports the high-party-count sweeps for Kreyvium/Boolean and Pasta/BFV. Section 6.6 closes with resource costs and measurement specifics. The interpretation and discussion of these results is given in Chapter 7.

6.1. Security Analysis

All pipelines are compared at a common target security level of $\lambda = 128$ bits [23]. This is necessary for a fair comparison: a pipeline should not obtain lower latency, smaller ciphertexts, or a deeper circuit by using weaker parameters. For the TFHE backends, the Boolean and shortint configurations use parameter sets targeting this security level. For BFV, the ring degree and modulus chain must also remain within a 128-bit secure parameter budget.

The experiments follow the honest-but-curious model of Section 4.2: each transciphering pipeline recovers the same FHE-encrypted Bloom-filter bits the direct-FHE baseline starts from, after which the identical threshold circuit runs.

The evaluation does not attempt to cryptanalyse the symmetric ciphers. AES-128 is included as a standardised anchor [32]. Kreyvium, Grain-128a, Rasta, and Pasta are included because they are relevant to FHE-friendly transciphering [18, 34, 35, 43]. The question measured here is therefore not whether each cipher is preferable in ordinary software, but how costly its decryption circuit becomes under the selected FHE backend.

The BFV backend introduces one additional security condition. Unlike the TFHE backends, the BFV pipeline is evaluated in leveled mode. It therefore has a finite multiplicative-depth budget, determined by the modulus chain that remains secure for the selected ring degree. The direct-BFV baseline needs to fit only the threshold circuit. The Pasta/BFV transciphering pipeline must fit the Pasta decryption circuit followed by the threshold circuit. If no 128-bit-secure BFV parameter set can support the required chain, the run fails instead of silently lowering the security level. This condition is central to the high-N results in Section 6.5.

Correctness is checked empirically against the plaintext oracle. For every completed FHE row, the direct-FHE result matches the plaintext threshold result on the sampled prefix: the first n bit-positions of the filter. For every completed transciphering row, the homomorphically decrypted bits match the original Bloom-filter bits, and the threshold result after transciphering matches the plaintext threshold

result. The false positives reported in this chapter are therefore not encryption errors. They are the expected false positives of the Bloom-filter threshold construction.

6.2. Experimental Methodology and Metrics

A fair comparison needs one fixed set of metrics, applied to every pipeline under the same conditions. The metrics fall into three groups: server compute, communication, and accuracy. Together, these answer the research questions of Section 1.4: server compute (RQ1), upload bandwidth (RQ2), the deployment-dependent pipeline comparison (RQ3), protocol scaling (RQ4), algebra/backend matching (RQ5), and the Bloom-filter accuracy trade-off (RQ6).

6.2.1. Experimental Setup

The experiments were run as DelftBlue HPC jobs using a fixed 16-core configuration. The implementation was compiled with native CPU optimisations and a fixed Rayon thread count. This makes the wall-clock results comparable within the campaign, although the absolute timings should still be read as measurements on this hardware rather than hardware-independent constants.

The plaintext protocol sweeps are run at full scale, because they do not require FHE. The FHE sweeps are measured on a sampled prefix of the Bloom filter and extrapolated to the full filter. This is necessary because some pipelines would take months or years if executed over a full $S = 10^6$ filter. The extrapolation is justified by the structure of the workload: every Bloom-filter position evaluates the same circuit shape, so the cost per position does not depend on which position is being evaluated. The measured per-bit cost is therefore the headline figure, while the full-filter values are estimates derived by multiplying this per-bit cost by the full Bloom-filter length.

Each pipeline's cap is the largest sampled prefix that still fills the cipher's block (or the SIMD lanes), giving a steady-state (the per-bit cost once each cipher's block or SIMD lanes are fully filled) per-bit cost, while completing within the HPC wall-time budget. The configured cap is 8192 bits. Faster ciphers reach it and slower ones are reduced, the values being tuned empirically against the wall-time budget. The sampled prefix differs between pipelines. AES/Boolean and AES/shortint use a 512-bit cap. Kreyvium uses an 8192-bit cap in the set-size sweep. Grain is reported using the completed 512-bit rerun, denoted `grain128a_sb512`. The original 8192-bit Grain run is partial, due to it running against its allocated wall-time. Rasta/shortint is reported from the 512-bit run. Its 128-bit rerun under-fills Rasta's 525-bit block and is therefore not used as the representative per-bit result. Pasta/BFV uses a SIMD batch once the set size is large enough. The smallest set-size rows under-fill the SIMD lanes, so they are not used as steady-state per-bit values.

The full FHE parameter sets, sample-bit caps, and the hardware configuration are listed in Appendix A.

6.2.2. Datasets and Parameters

The workload is synthetic threshold PSI over IPv4-like identifiers. Each party holds a set of size S. A fixed number C of common IPs is planted across the parties, and the remaining elements are sampled so that the exact threshold result is known. Each party's set is encoded as a Bloom filter. The Bloom-filter length M and number of hash functions K are determined by the set size and the per-filter false-positive target, which the implementation sets to $\varepsilon = 1/S$. This sizes the filter at about $1.44 \log_2 S$ bits per element (28.76 bits per element at $S = 10^6$), which is what produces the filter lengths of Table 6.1. Double hashing is used to derive the Bloom-filter positions efficiently [65]. The datasets are generated by a deterministic RNG (a fixed seed, `0xBEEF0042`), so every run is byte-identical and the plaintext result is an exact oracle. Generation proceeds in two steps. First, C distinct random IPv4 addresses are drawn as the common items, and each is assigned to a uniformly random subset of exactly T of the N parties: the planted threshold result. Second, each party's set is filled up to size S with fresh random IPv4 addresses that are globally unique (tracked against a running universe set, so the non-common elements are disjoint across parties and no duplicates occur). The accuracy metric evaluates the union of all N party sets. Generated sets are cached on disk and reloaded unchanged on repeated runs.

Four one-axis plaintext sweeps isolate the protocol parameters:

- **Party sweep:** $N = 2, \ldots, 20$, with $T = \lceil N/2 \rceil$, $S = 10^6$, and $C = 10$.

- **Threshold sweep:** $N = 20$, $T = 1, \ldots, 20$, $S = 10^6$, and $C = 10$.
- **Set-size sweep:** $S \in \{10, 100, 10^3, 10^4, 10^5, 10^6\}$, with $N = 3$, $T = 2$, and $C = 10$.
- **Common-IP sweep:** $C \in \{10, 100, 10^3, 10^4, 10^5\}$, with $N = 20$, $T = 10$, and $S = 10^6$.

The ranges bracket realistic deployments while exposing the asymptotics: set size spans five orders of magnitude to $S = 10^6$ (a large per-organisation indicator set), party count runs to $N = 20$ (beyond which the $\binom{N}{T}$ circuit is already intractable on the per-gate backends), and C reaches 10^5, enough to separate the false-positive background from the true signal.

The FHE set-size comparison uses the same $N = 3$, $T = 2$, $C = 10$ setting as the plaintext set-size sweep. This is the smallest non-trivial majority threshold and gives a controlled point at which all pipelines can be compared. The high-N FHE comparison uses two representative pipelines, Kreyvium/Boolean and Pasta/BFV, over $N = 2, 4, 6, 8, 10, 12$ with $T = \lceil N/2 \rceil$.

6.2.3. Server Compute

The compute cost is the work the server does. It is reported in two ways.

The first is wall-clock latency, broken down per stage of the pipeline: FHE encryption, direct-FHE SOP evaluation, FHE decryption, transciphering setup, homomorphic symmetric decryption, SOP evaluation after transciphering, and final decryption. This breakdown matters because the stages scale differently. Setup is fixed. Homomorphic symmetric decryption scales with the number of encrypted payload bits. The SOP stage scales with both the Bloom-filter length and the threshold circuit size.

The second is the per-bit transciphering cost, reported in milliseconds per bit. This isolates the additional server work introduced by transciphering: the homomorphic evaluation of the symmetric decryption circuit. For the TFHE Boolean backend, this cost follows the number of bootstrapped Boolean gates. The headline nonlinear-operation count per output bit (Table 5.2) is useful for understanding the ordering of ciphers, but it is not a literal wall-clock multiplier: on the Boolean backend the realised bootstrap count is the cipher's total gate count, because XOR and OR gates are bootstrapped as well as AND gates. For shortint, XOR-heavy linear work can be represented more cheaply, while nonlinear work still requires programmable bootstrapping. For BFV, the limiting resource is multiplicative depth and SIMD utilisation rather than a per-gate bootstrap count [15, 13].

Unless otherwise stated, each FHE timing row is a single wall-clock measurement on the fixed DelftBlue configuration. The full-filter values are extrapolated from sampled prefixes. The timing results should therefore be read as comparative measurements within one campaign, not as statistically averaged hardware-independent performance constants.

6.2.4. Communication

The communication cost is the size of the per-party upload. The direct-FHE baseline uploads every Bloom-filter bit as an FHE ciphertext. The transciphered pipeline uploads a symmetric ciphertext and one FHE-encrypted symmetric key. Communication is reported as an absolute upload size and as ciphertext expansion:

$$\text{expansion} = \frac{\text{uploaded bytes}}{\text{plaintext bytes}}.$$

The bandwidth saving is the direct-FHE upload divided by the transciphered upload. This ratio is useful, but must be interpreted together with the absolute upload size. A pipeline can have a large saving ratio because its direct-FHE baseline is especially large. For that reason, the evaluation reports both expansion and absolute upload.

6.2.5. Accuracy

Accuracy is a property of the TPSI workload rather than of any particular FHE backend. Once correctness against the plaintext oracle has been established, the encrypted and plaintext pipelines produce the same threshold result on the evaluated prefix. The remaining error is the Bloom-filter false-positive behaviour. Accuracy is therefore reported as the number of true threshold items, the number of false

Table 6.1: Protocol accuracy in the set-size sweep ($N = 3$, $T = 2$, $C = 10$).

S	Bloom bits M	True threshold items	False positives	Observed FPR
10	48	10	2	0.200
100	959	10	46	0.164
10^3	14,378	10	166	0.0557
10^4	191,702	10	595	0.0198
10^5	2,396,265	10	2,667	0.00889
10^6	28,755,176	10	9,868	0.00329

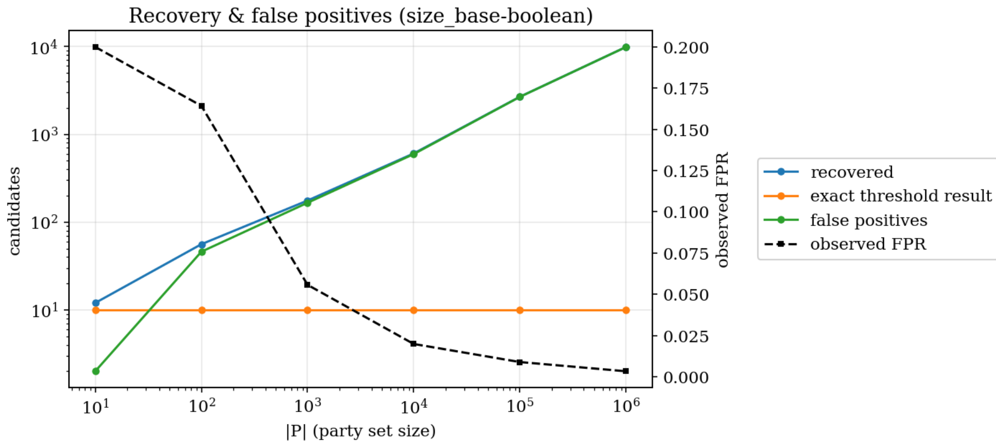


Figure 6.1: Recovery and false positives as the party set size grows. The true threshold result is recovered in every row, while the false-positive rate falls but the absolute number of false positives grows.

positives, and the observed false-positive rate:

$$\mathrm{FPR}_{obs} = \frac{\text{false positives}}{\text{candidate pool size} - \text{true threshold items}},$$

where the candidate pool is the set of items tested for membership against the result filter, namely the union of all parties' sets, so the denominator counts only the non-member candidates that can become false positives. The correction matters only at the smallest scale: at $S = 10$ the pool holds 20 candidates of which 10 are true, so the two false positives give $\mathrm{FPR}_{obs} = 2/(20 - 10) = 0.200$. For large S the true items are a negligible fraction of the pool and the correction vanishes. No false negatives were observed in the completed sweeps: the planted threshold result was always recovered. The accuracy results therefore concern precision rather than recall.

6.3. Protocol Scaling

The plaintext sweeps characterise the TPSI protocol independently of FHE. They show how the Bloom-filter encoding and the threshold circuit behave before any transciphering cost is introduced.

6.3.1. Scaling with Set Size

The set-size sweep fixes $N = 3$, $T = 2$, and $C = 10$, and varies S from 10 to 10^6. The Bloom filter is resized as the set grows. At $S = 10^6$, the filter has $28{,}755{,}176$ bits and uses 20 hash functions, or approximately 28.76 bits per element.

The planted threshold result contains 10 items and is recovered in every row. The false-positive rate falls from 0.20 at $S = 10$ to 0.00329 at $S = 10^6$. The absolute number of false positives nevertheless grows, from 2 to 9,868, because the candidate pool grows much faster than the rate falls. Figure 6.1 shows the same behaviour. The important point is that the protocol-level false-positive rate is not the same as the per-filter target used to size the Bloom filter. The 2-of-3 threshold circuit amplifies the false-positive behaviour of the individual filters.

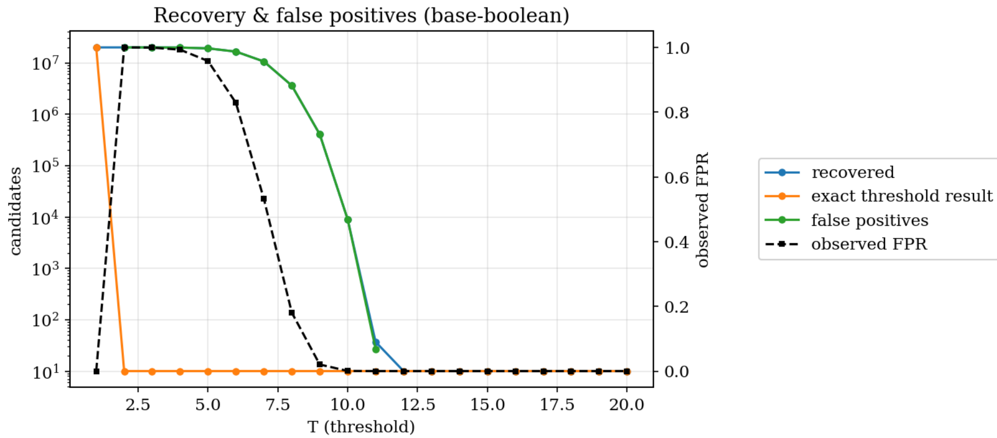


Figure 6.2: Recovery and false positives in the threshold sweep ($N = 20$, $S = 10^6$, $C = 10$). Low thresholds are too permissive, while false positives collapse around the upper-middle thresholds.

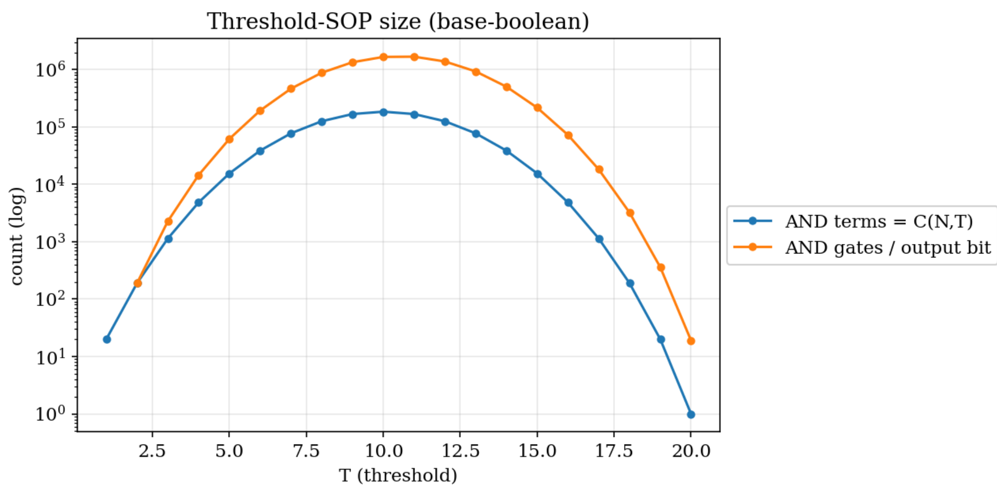


Figure 6.3: Threshold-SOP size in the threshold sweep. The number of terms peaks near the middle threshold, which is also the region where false positives become usable.

6.3.2. Scaling with the Threshold

The threshold sweep fixes $N = 20$, $S = 10^6$, and $C = 10$, and varies T from 1 to 20. This sweep shows that the threshold is the dominant accuracy knob.

At low thresholds, the output is almost entirely false positives. At $T = 2$, nearly every candidate qualifies, producing 19,999,268 false positives and an observed FPR of 0.99996. The FPR then collapses through a knee between $T = 7$ and $T = 11$. At $T = 10$, 8,976 false positives remain. At $T = 11$, only 27 remain. From $T = 12$ onward, no false positives are observed.

The threshold also determines the circuit size. A single output bit is an OR over $\binom{N}{T}$ products of T party bits. For $N = 20$, the term count peaks at $T = 10$ with $184{,}756$ terms. The AND-gate count peaks around the same region, because each term also becomes longer as T grows. The useful accuracy region therefore overlaps with the most expensive part of the SOP circuit.

6.3.3. Scaling with the Number of Parties

The party sweep fixes $S = 10^6$ and $C = 10$, and sets $T = \lceil N/2 \rceil$. It shows both the combinatorial growth of the threshold circuit and an accuracy pattern caused by the threshold ratio.

The circuit grows rapidly with N. At $N = 3$, $T = 2$, the circuit has 3 product terms and approximately 5 total Boolean gates per output bit. At $N = 10$, $T = 5$, it has 252 terms and about 1,259 gates per bit.

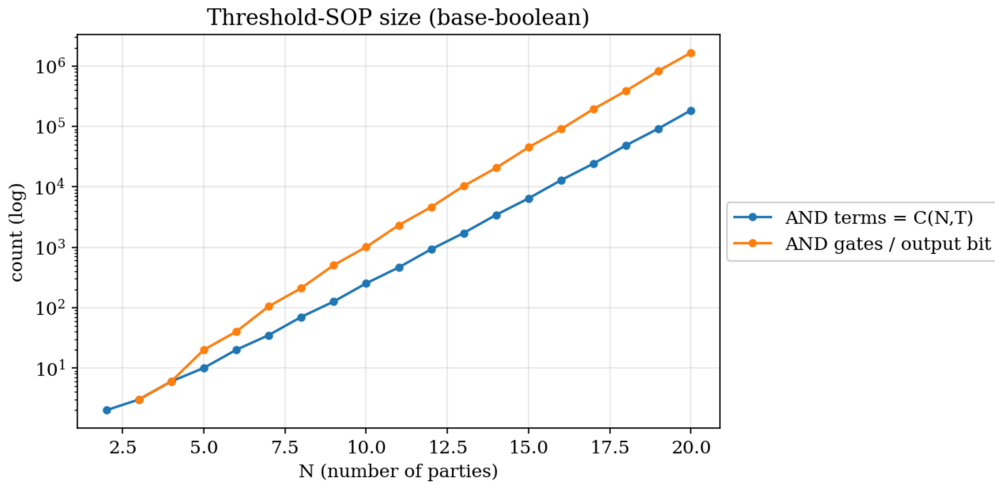


Figure 6.4: Threshold-SOP size as the number of parties grows with $T = \lceil N/2 \rceil$. The circuit grows combinatorially, making the threshold circuit itself a major cost driver.

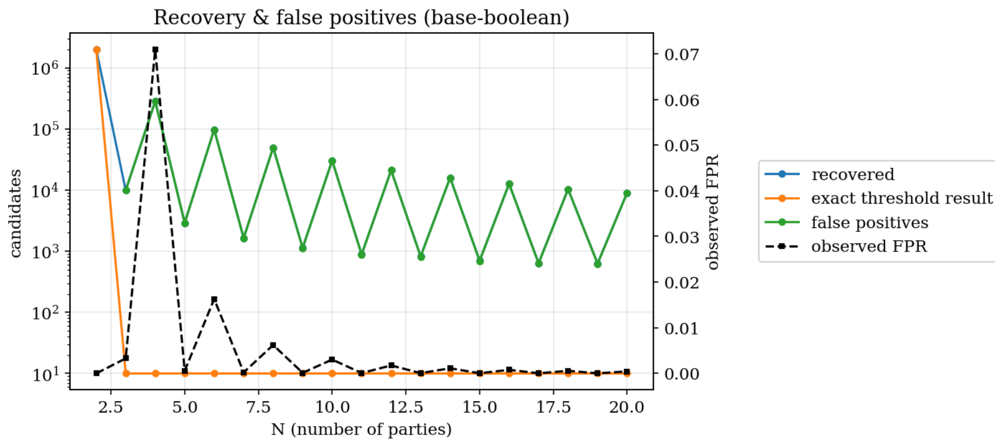


Figure 6.5: Recovery and false positives as the number of parties grows with $T = \lceil N/2 \rceil$. The false-positive rate follows an even/odd sawtooth caused by the threshold ratio.

At $N = 20$, $T = 10$, it reaches 184,756 terms, 1,662,804 AND gates, and 184,755 OR gates per output bit. On the Boolean backend, where every gate is bootstrapped, this gate count is directly tied to the direct-FHE SOP cost.

The accuracy does not fall smoothly with N. Instead, the party sweep splits into two branches. For even N, the policy $T = \lceil N/2 \rceil$ gives exactly $T/N = 0.5$. For odd N, it gives a ratio strictly above one half. The odd branch has substantially lower FPR than the even branch at comparable party counts. This means that precision is governed by the threshold ratio, not by the number of parties alone. Adding parties does not automatically improve precision. Choosing a threshold just above half does.

6.3.4. Scaling with the Number of Common Items

The common-IP sweep fixes $N = 20$, $T = 10$, and $S = 10^6$, and varies the planted common count C. The exact planted result is recovered in every row. The false-positive count stays roughly fixed around 8,000 to 9,000 for $C \leq 10^4$, then falls to 3,119 at $C = 10^5$ (Table 6.2).

This shows that C is primarily a signal-size parameter. It controls how many true items exist in the result, but it does not fundamentally change the false-positive mechanism. Precision improves as C grows because the true signal becomes large relative to the roughly fixed false-positive background. Figure 6.6 shows the same behaviour.

Table 6.2: Common-IP sweep ($N = 20$, $T = 10$, $S = 10^6$).

C	True threshold items	False positives	Observed FPR
10	10	8,976	$4.49 \cdot 10^{-4}$
100	100	8,982	$4.49 \cdot 10^{-4}$
10^3	1,000	8,800	$4.40 \cdot 10^{-4}$
10^4	10,000	8,088	$4.06 \cdot 10^{-4}$
10^5	100,000	3,119	$1.63 \cdot 10^{-4}$

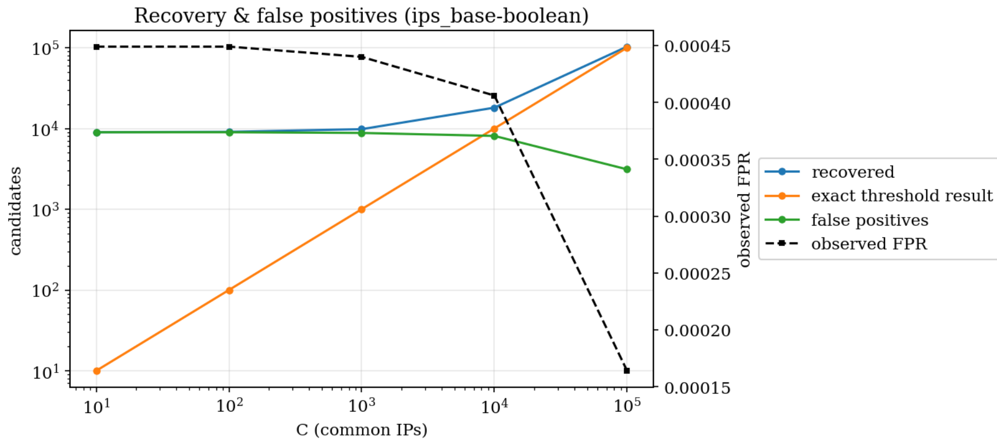


Figure 6.6: Recovery and false positives in the common-IP sweep ($N = 20$, $T = 10$, $S = 10^6$). The exact planted result is recovered in every row. The false-positive background stays roughly fixed until the true signal grows large enough to dominate it.

6.4. Pipeline Comparison

The pipeline comparison uses the set-size sweep at $N = 3$, $T = 2$, and $C = 10$. This setting is used to compare every completed direct-FHE and transciphering pipeline on the same workload.

6.4.1. Per-Bit Transciphering Cost

Figure 6.7 shows the measured per-bit transciphering cost. The steady-state ranking is clear:

$$\text{Pasta/BFV} \ll \text{Kreyvium/Boolean} < \text{Grain/Boolean} < \text{AES/Boolean} \approx \text{AES/shortint} \ll \text{Rasta/shortint}.$$

Table 6.3 gives the representative large-input values. Pasta/BFV is the fastest measured pipeline once the SIMD lanes are filled, stabilising around 4.1 ms/bit. Kreyvium/Boolean is the fastest Boolean pipeline at about 313 ms/bit. Grain/Boolean is about 3.5 times slower than Kreyvium. AES/Boolean and AES/shortint are both around four to five seconds per bit. Rasta/shortint is the slowest measured pipeline, at around 18.5 seconds per bit. The full-filter times in Table 6.3 are extrapolations from the sampled per-bit cost, not measured end-to-end runtimes.

The stage breakdowns (Figure 6.8) confirm that the homomorphic symmetric-decryption stage dominates the sampled latency for the slower ciphers. For AES and Rasta, this stage is the overwhelming cost. For Kreyvium and Grain, the cost is lower but still large in absolute terms. For Pasta/BFV, the sampled homomorphic decryption is much smaller because the BFV backend evaluates packed prime-field slots.

6.4.2. Upload Bandwidth

The bandwidth comparison shows the benefit transciphering was designed to provide. At $S = 10^6$, the plaintext Bloom-filter payload is 3.43 MiB. Direct Boolean FHE expands this to 87.30 GiB. Boolean transciphering reduces the upload to 3.83 MiB, an expansion of only $1.116\times$ and a saving of $23{,}367\times$ over direct Boolean FHE.

The saving appears only after the fixed FHE-encrypted key is amortised. At $S = 10$, transciphering is

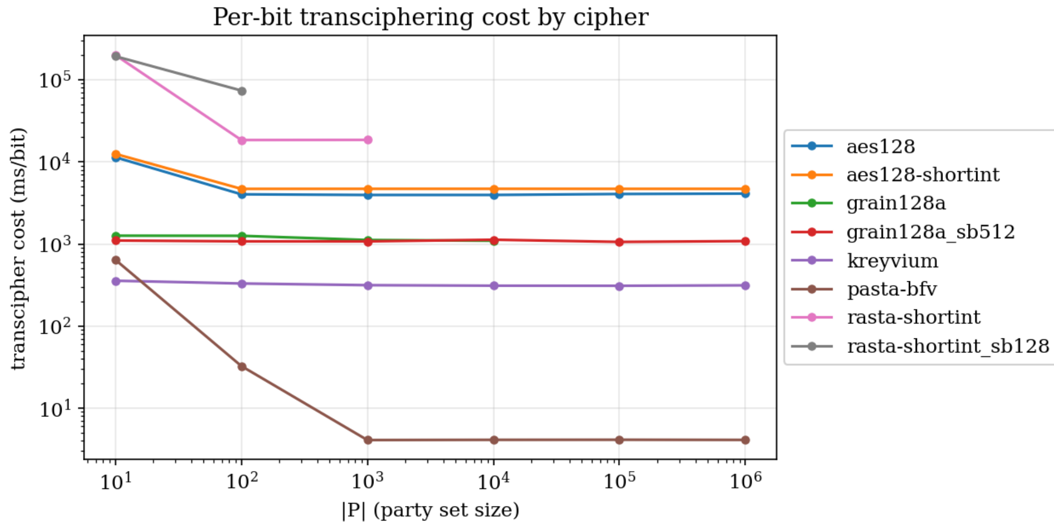


Figure 6.7: Per-bit homomorphic symmetric-decryption cost by pipeline. Small-input rows under-fill the relevant block or SIMD structure for some ciphers; the steady-state comparison uses the filled rows.

Table 6.3: Representative pipeline comparison at $S = 10^6$, $N = 3$, $T = 2$, $C = 10$. The full-filter time estimates refer to the homomorphic symmetric-decryption stage per party, not to the complete pipeline including the threshold SOP.

Pipeline	TC cost	Full TC decrypt est.	TC upload	TC expansion
Pasta/BFV	4.11 ms/bit	32.8 h	58.8 MiB	17.16×
Kreyvium/Boolean	313 ms/bit	104 d	3.83 MiB	1.116×
Grain/Boolean	1,083 ms/bit	360 d	3.83 MiB	1.116×
AES/Boolean	4,107 ms/bit	3.74 y	3.83 MiB	1.116×
AES/shortint	4,695 ms/bit	4.28 y	5.44 MiB	1.586×
Rasta/shortint	18,500 ms/bit	∼ 17 y	omitted[a]	omitted[a]

[a] Rasta/shortint upload and expansion are omitted because the completed runs do not provide a representative full-filter bandwidth measurement. They are therefore excluded from the bandwidth comparison to avoid mixing completed and partial values.

larger than direct Boolean FHE because the payload is only a few dozen bits. By $S = 100$, it already gives a positive saving. At $S = 10^6$, the upload is almost plaintext-sized.

Pasta/BFV is different. Direct BFV is already more compact than direct Boolean FHE because BFV packs values into SIMD slots [66]. At $S = 10^6$, direct BFV upload is about 943 MiB, and Pasta/BFV transciphering reduces this to 58.8 MiB. This is a real saving, but only about $16\times$ over direct BFV. Pasta/BFV also has a larger absolute transciphered upload than the TFHE stream-cipher pipelines because each bit is represented as a prime-field word. In expansion terms this makes Pasta/BFV the least bandwidth-efficient pipeline measured, $17.16\times$ against the $1.116\times$ of the Boolean stream ciphers, so its compute advantage comes with the largest transciphered upload.

6.4.3. Direct-FHE SOP Baselines

The direct-FHE columns measure the cost of evaluating the threshold SOP without transciphering. At $N = 3$, $T = 2$, and $S = 10^6$, the full-filter direct Boolean SOP estimate is about 34.6 days. This figure is a single measurement. Across the three Boolean runs the same estimate ranges from about 33 to 35 days, roughly five percent, so it should be read as one realisation rather than an exact constant. The shortint baseline is of the same order, around 40 days. The direct-BFV SOP estimate is about 508 seconds, or 8.5 minutes, because the SOP is evaluated over packed SIMD slots.

These direct baselines matter because transciphering does not remove the threshold circuit. It changes the upload and adds a homomorphic decryption stage before the same TPSI computation. The SOP cost is therefore shared by direct and transciphered pipelines on the same backend. This becomes important in the high-party-count sweeps.

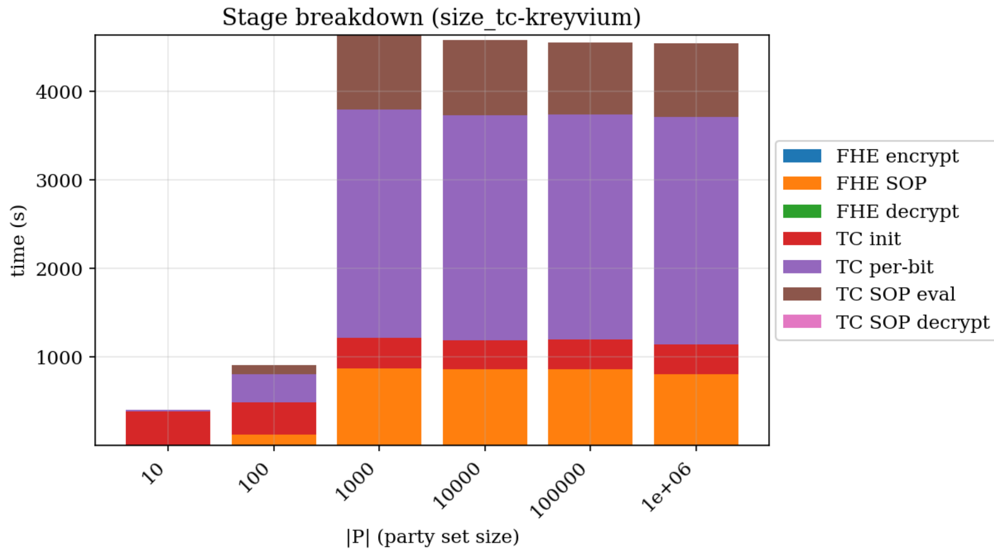


Figure 6.8: Per-stage latency breakdown for Kreyvium/Boolean across the set-size sweep ($N = 3$, $T = 2$). The homomorphic symmetric-decryption stage dominates the sampled cost; the threshold SOP is a small share at this party count.

Per-pipeline stage breakdowns and latency curves are given in Appendix C.

6.5. BFV at High N

The set-size comparison fixes $N = 3$. To measure how the pipelines behave when the threshold circuit grows, two high-N FHE sweeps were run: Kreyvium/Boolean and Pasta/BFV. Both use $T = \lceil N/2 \rceil$ and vary N from 2 to 12.

6.5.1. Kreyvium/Boolean

In the Kreyvium/Boolean party sweep, the homomorphic Kreyvium decrypt stage remains almost constant as N grows. This is expected, because the Kreyvium keystream cost depends on the number of bits and the cipher, not on the number of parties in the threshold circuit. The sampled transciphering stage stays around 308–350 ms/bit.

The SOP stage grows with the threshold circuit. On the 64-bit sample, SOP evaluation takes about 1.6 seconds at $N = 2$, 337 seconds at $N = 8$, 1,750 seconds at $N = 10$, and 7,361 seconds at $N = 12$. The SOP overtakes the cipher-decrypt stage around $N = 8$ and dominates it by $N = 12$.

This sweep also validates the Boolean per-gate cost model across larger circuits. Dividing the SOP time by the number of gates and evaluated bits gives approximately 0.019–0.025 seconds per bootstrapped gate across the measured range. Extrapolated to the $N = 20, T = 10$ protocol setting, where the SOP has about 1.85 million gates per output bit, this implies approximately 10.7 hours per output bit and roughly 35,000 years for a full $S = 10^6$ filter. This is not a practical target, as it shows that a per-gate-bootstrapped Boolean SOP is not the right high-N strategy.

6.5.2. Pasta/BFV

The Pasta/BFV high-N sweep gives two separate results: the direct-BFV SOP baseline and the Pasta/BFV transciphered path.

The direct-BFV SOP baseline remains measurable through $N = 12$. The sampled SOP time grows from 0.015 seconds at N = 2 to 2,926 seconds at N = 12. The ring degree, and with it the SIMD batch, grows with the circuit depth: 8,192 slots at N = 2, 16,384 at N = 4, and 32,768 from N = 6 onward. At $N = 12$, this is about 0.089 seconds per bit. Compared with the Boolean Kreyvium SOP at the same N, this is roughly a 1,300-fold per-bit advantage. BFV's SIMD packing is therefore the only measured path that keeps the SOP itself within a plausible range at higher party counts.

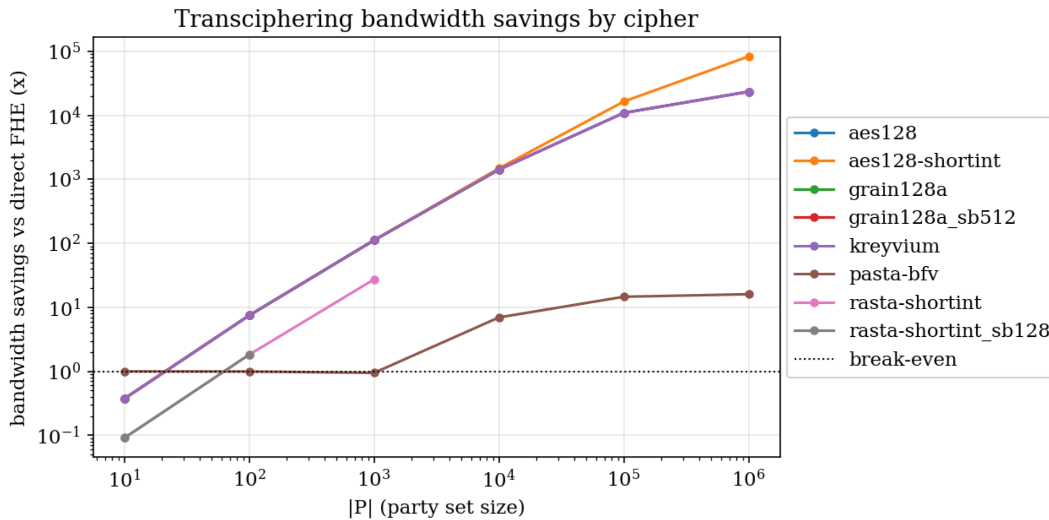


Figure 6.9: Bandwidth saving of transciphering over each pipeline's direct-FHE baseline. The ratio should be read together with the absolute upload size, because a large saving can also come from a very large direct-FHE baseline.

The Pasta/BFV transciphered path does not remain securable over the same range. It runs at $N = 2$ and $N = 4$, but from $N = 6$ onward no 128-bit-secure SIMD degree supports the required modulus chain. The direct-BFV baseline fits the SOP alone to $N = 12$, but the transciphered path must fit the Pasta decryption circuit before the SOP. The added depth consumes the available BFV modulus budget earlier.

This result is a security-ceiling result rather than a runtime timeout. For the bootstrapped TFHE backends, noise is refreshed by bootstrapping and this kind of depth ceiling is not the limiting factor. For leveled BFV, the remaining noise budget after transciphering is a binding resource. Transciphering therefore narrows the feasible high-N range of the BFV pipeline relative to direct BFV.

6.6. Resource Cost

The memory profile separates the TFHE and BFV backends. The Boolean and shortint pipelines remain in the low-memory range in the set-size sweeps, roughly a few hundred megabytes. In the high-N Kreyvium sweep, memory rises with the larger SOP circuit but remains far below the BFV context. Pasta/BFV uses roughly 17 GB in the set-size sweep and around 19 GB in the high-N sweep once the BFV context is fully allocated. This memory cost is largely fixed by the BFV parameters and appears even for small inputs.

The evaluation also has several measurement caveats. First, full-filter FHE times are extrapolated from sampled prefixes. This is appropriate for these bit-independent circuits, but the full-filter figures should be read as estimates, while the sampled timings are direct measurements. Second, small rows can under-fill the relevant computation unit. AES under-fills its 128-bit block at $S = 10$. Pasta under-fills the BFV SIMD lanes at $S = 10$ and $S = 100$. The Rasta 128-bit rerun under-fills the 525-bit Rasta block. Third, the results are measured on one 16-core HPC configuration. They compare the pipelines under identical conditions, but they are not claimed to be production-optimised hardware ceilings.

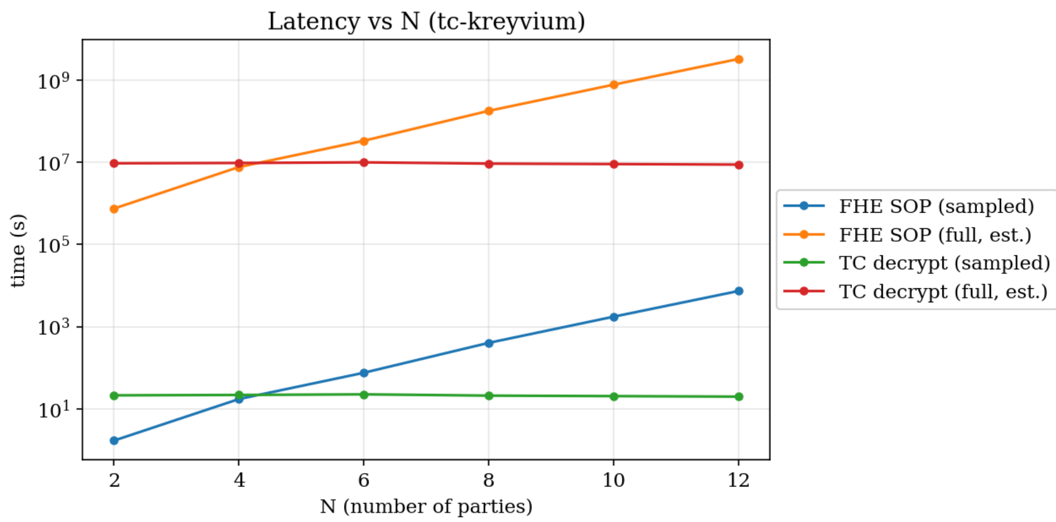


Figure 6.10: Latency in the Kreyvium/Boolean high-N sweep. The transciphering cost is almost independent of N, while the SOP grows with the threshold circuit.

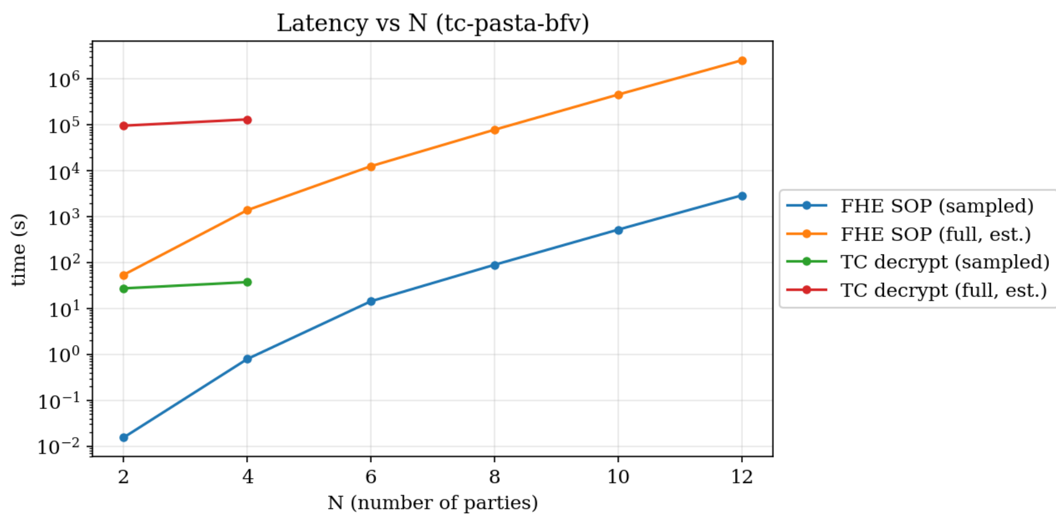


Figure 6.11: Latency in the Pasta/BFV high-N sweep. Direct-BFV SOP timings continue to $N = 12$, while the Pasta/BFV transciphering path is only securable to $N = 4$ under the 128-bit BFV parameter constraint.

7
Discussion and Conclusion

Chapter 6 reported the measured behaviour of the TPSI protocol and the transciphering pipelines. This chapter interprets those results in terms of the research questions. Section 7.1 explains the main findings: the deployment-dependent trade-off, the influence of cipher/backend matching, the high-N SOP bottleneck, and the BFV security ceiling. Section 7.2 states the limitations of the evaluation. Section 7.3 gives directions for future work. Section 7.4 concludes the thesis by answering the research questions.

7.1. Interpretation of Findings

The central result is that transciphering is a bandwidth/server-compute trade-off, and that the best pipeline depends on the deployment. Transciphering is highly effective at reducing upload bandwidth. In the Boolean pipelines, direct FHE expands the large $S = 10^6$ Bloom-filter upload to 87.30 GiB per party, while transciphering reduces it to 3.83 MiB (Figure 6.9). This is essentially the plaintext Bloom-filter size plus an amortised encrypted key. The cost of this reduction is that the server must homomorphically evaluate a symmetric decryption circuit before it can run the TPSI threshold circuit.

7.1.1. No Single Pipeline Wins

The measured pipelines form a Pareto outcome rather than a total ordering. Pasta/BFV is the minimal compute point in the low-party configuration. Once the SIMD lanes are filled, it costs about 4.1 ms per bit for the homomorphic Pasta decryption stage, about 76 times faster than Kreyvium/Boolean and about 1000 times faster than AES/Boolean (Figure 6.7, Table 6.3). This makes it the natural choice when server compute is the limiting resource, the server has enough memory, and the party count is low enough to remain inside the BFV transcipher security ceiling.

Kreyvium/Boolean is the minimal bandwidth point. It keeps the Boolean transciphering upload close to the plaintext size and is the fastest measured Boolean transciphering pipeline. It is still much slower than Pasta/BFV in server compute, but it uses far less memory and far less upload. It is therefore the better choice when upload bandwidth is the limiting resource and the server-side work can be tolerated or parallelised.

The other pipelines explain why the comparison has to be pipeline-level. Grain is far cheaper than AES under FHE, but in this workload it is slower than Kreyvium while giving the same Boolean upload. AES is fast on ordinary hardware but expensive under FHE, and therefore functions as a standardised anchor rather than a recommendation. AES/shortint shows that moving a cipher to a different backend does not help unless the cipher's expensive operations match what that backend makes cheap. Rasta/shortint shows the same point more strongly: shortint is the right backend for Rasta compared with Boolean Rasta, because it avoids bootstrapping dense affine layers as Boolean XOR gates, but the measured Rasta/shortint pipeline is still the most expensive one.

The answer to RQ3 is therefore not simply to choose a cipher. It is a decision based on the scenario. For bandwidth-constrained deployments, the Kreyvium/Boolean corner is the strongest measured op-

Table 7.1: Choosing a pipeline by deployment setting. The decision rests on identifying the binding resource for the deployment, then reading across the row.

Binding constraint	Recommended pipeline	Why	Trade-off
Upload bandwidth, large inputs	Kreyvium / Boolean	Near-plaintext upload ($1.12\times$); cheapest Boolean cost, ≈ 313 ms/bit	High server compute; the SOP grows with N
Server compute, low N, ample RAM	Pasta / BFV	Fastest per bit ($\approx$ 4 ms); SIMD throughput; compact direct baseline	$\approx$ 17 GB RAM; largest upload ($17\times$ expansion); transcipher securable only to $N \approx 4$
High party count, N large	Direct BFV (no transciphering)	SIMD keeps the threshold SOP tractable where Boolean cannot	Transcipher saving is lost; BFV transcipher hits the ceiling
Standardised cipher required	AES / Boolean	Drop-in standard, correctness-anchored	Expensive, $\approx$ 4–5 s/bit; an anchor, not a recommendation

tion. For compute-constrained, low-party deployments with sufficient RAM, the Pasta/BFV corner is the strongest measured option. For high-party deployments, neither answer is complete, because the threshold circuit itself becomes the limiting factor. Table 7.1 summarises the decision by binding constraint.

7.1.2. Algebra/Backend Match Decides Server Cost

The per-bit cost ranking confirms the match hypothesis, with one caveat about what 'a cipher's algebra' means. A cipher is cheap to transcipher only when its dominant operations are the ones the backend makes cheap, and that is decided by its operational structure, the balance of its linear and nonlinear layers, not by the algebraic domain it is defined over. Sharing the domain is necessary but not sufficient, as Rasta shows below.

On the Boolean backend, every gate is bootstrapped. This makes total Boolean gate count the relevant cost, not only nonlinear count. Kreyvium is the best fit among the measured Boolean ciphers because each keystream bit has a small Boolean footprint and low depth. Grain is still FHE-friendly relative to AES, but its larger Boolean footprint makes it about 3.5 times slower than Kreyvium (Figure 6.7). AES is much worse because its S-box-heavy structure is not designed for FHE evaluation.

Rasta also cautions how that match should be judged. By shared algebraic domain it looks like the ideal Boolean cipher: it is defined over GF(2) and the workload is bit-valued, so cipher, workload, and backend agree on this domain. Yet Boolean Rasta is the worst case in the comparison, because the domain a cipher is defined over says nothing about which of its operations dominate the circuit. Rasta spends its cost on the wide XOR of its affine layers, and the Boolean backend charges a bootstrap for every XOR, so the apparent fit is the wrong reading. The deciding factor is not shared algebra but whether the backend makes the cipher's *dominant* operation cheap.

On shortint, XOR-heavy work can be represented more cheaply, so shortint can rescue designs that would be especially bad on Boolean. That is why Rasta/shortint is meaningful as a comparison point. The size of the rescue can be bounded from the Boolean cost model: at the measured per-gate cost of roughly 0.02 s (Section 6.5), Rasta's approximately 1,500 bootstrapped gates per bit imply on the order of 30 s per output bit on Boolean, against the measured 18.5 s per bit on shortint. That is a saving of less than a factor of two in wall-clock, despite the hundredfold drop in bootstrap count. However, the rescue is not enough to make it competitive. Even with its XORs leveled, the sheer volume of its affine work, on the order of 1,500 leveled additions per output bit, together with its remaining nonlinear layers, keeps its measured per-bit cost the largest in the comparison (Table 6.3). The conclusion is not that shortint is useless, but that matching must be specific: a backend may rescue one component of a

cipher while the resulting pipeline is still a poor choice.

On BFV, Pasta is the best match because it is built over a prime field and maps naturally to BFV's SIMD arithmetic. BFV does not pay one bootstrap per Boolean gate. It pays in multiplicative depth and parameter size, while applying operations to many slots at once. This is why Pasta/BFV is the throughput point. The same property also explains why BFV evaluates the SOP much faster than Boolean at high N.

7.1.3. The Protocol Dominates at High Party Counts

The high-N sweeps change the interpretation of transciphering. At $N = 3$, it is reasonable to focus on the symmetric decryption circuit, because the threshold circuit is still small. At larger N, the SOP grows combinatorially. For $N = 20$, $T = 10$, the Boolean circuit has about 1.85 million gates per output bit (Figure 6.4). At the measured Boolean per-gate cost, this corresponds to hours per output bit and tens of thousands of years for a full $S = 10^6$ filter. No choice of symmetric cipher fixes this, because the SOP is the application circuit that remains after transciphering.

This means that transciphering is not a complete high-N solution for bitwise TPSI. It removes the upload expansion and can make the cipher-decryption part cheaper, but it does not change the threshold circuit. In the Kreyvium high-N sweep, the SOP overtakes the cipher-decrypt stage around $N = 8$ and dominates by $N = 12$ (Figure 6.10). From that point onward, improving the symmetric cipher gives diminishing returns: the application circuit is the wall.

BFV's SIMD packing changes the high-N picture. It evaluates the SOP over many slots in parallel and is roughly three orders of magnitude faster than the Boolean SOP at $N = 12$ (Figure 6.11). This is why the high-N story becomes BFV-centric. However, BFV introduces its own limit.

7.1.4. The BFV Ceiling Is a Transciphering Cost

The BFV high-N sweep shows a cost of transciphering that is not captured by bandwidth or per-bit runtime alone. The direct-BFV baseline, which evaluates only the SOP, remains securable to $N = 12$ in the measured sweep. The Pasta/BFV transciphered path is only securable up to $N = 4$ (Figure 6.11). From $N = 6$ onward, the Pasta decryption depth plus the SOP depth requires a modulus chain that no longer fits the 128-bit secure BFV parameter budget.

This is due to the noise budget after transciphering. In a leveled scheme, homomorphic decryption consumes part of the depth budget that would otherwise be available for the application. The direct-FHE baseline has the whole budget available for the SOP. The transciphered BFV pipeline has already spent part of it before the SOP begins. Transciphering therefore narrows the feasible party count on BFV.

This does not contradict Pasta/BFV being the fastest low-N pipeline. It specifies the conditions in which that statement holds. Pasta/BFV is the throughput point when the combined decryption-plus-application circuit fits the leveled BFV budget. Once the party count pushes the SOP too deep, direct BFV can still run while Pasta/BFV transciphering cannot. The comparison therefore has three axes, not two: compute, bandwidth, and feasible depth/security.

7.1.5. Accuracy Is Governed by the Threshold Policy

The accuracy results are independent of the FHE backend because the encrypted results match the plaintext oracle. The relevant question is therefore the Bloom-filter threshold protocol itself.

The experiments show that recall is perfect in the measured sweeps: all planted threshold items are recovered. Precision is controlled by the threshold ratio and the Bloom-filter parameters. Low thresholds are too permissive and produce outputs dominated by false positives. In the $N = 20$ threshold sweep, the false-positive rate is almost one at $T = 2$, drops sharply around $T = 7$–$T = 11$, and becomes zero from $T = 12$ onward (Figure 6.2). The party sweep further shows that T/N matters more than N itself: $T = \lceil N/2 \rceil$ produces an even/odd sawtooth because even N gives exactly one half, while odd N gives a ratio just above one half (Figure 6.5).

This matters for deployment. A threshold just above half can suppress false positives much more strongly than a threshold exactly at half. However, middle thresholds are also where the SOP circuit

is largest (Figure 6.3). The protocol therefore contains an accuracy-cost trade-off before FHE is even considered.

7.2. Limitations

The limitations fall into two groups: how the measurements were taken, and how far they generalise.

The first limitation is that the FHE measurements use sampled prefixes and full-filter extrapolation. This is justified by the uniform per-bit circuit structure, and it is the only practical way to compare the slowest pipelines, but the multi-day and multi-year full-filter figures remain estimates. The sampled timings are measured wall-clock values. The full-filter values are derived from them.

Another measurement limitation is that plaintext/encrypted parity is checked only on the sampled prefix for the FHE runs. The plaintext sweeps run at full scale, and every completed FHE prefix matches the plaintext oracle, but the encrypted full filter is not executed for the slowest pipelines. This does not weaken the measured per-bit timing comparison, but it should be stated clearly.

The third measurement limitation is that the implementation is not a fully production-optimised HE implementation. Standard optimisations such as hoisting, batching refinements, more aggressive amortisation, or backend-specific circuit rewrites may change absolute runtimes. The conclusions most likely to survive such optimisation are the relative ones: Boolean per-gate bootstrapping makes large SOP circuits expensive, BFV SIMD gives much higher throughput, and cipher/backend algebra match strongly affects the transciphering stage.

On the generalisation side, one limitation is the selection of implemented ciphers. AES is standardised and therefore useful as an anchor. Kreyvium, Grain, Rasta, and Pasta are included because they are relevant to FHE-friendly transciphering, but their standardisation and cryptanalytic maturity differ. This thesis evaluates their homomorphic cost under the selected security assumptions. It does not attempt to settle their long-term suitability as general-purpose transport ciphers. Pasta-3 is the parameter set its designers propose for 128-bit security, which this thesis adopts for the cost comparison. The wider Rasta and Pasta family remains an active cryptanalysis target, and some related designs such as Agrasta have been broken [59], so its security margin is less settled than that of a standardised cipher like AES. A related point is implementation faithfulness. The cost of a pipeline is fixed by its circuit structure, the gate and operation counts and the multiplicative depth, which the implementations reproduce. AES is additionally validated against the FIPS-197 known-answer vectors. For Rasta and Pasta the public per-block layers are generated from a seeded pseudo-random generator rather than the reference specification's extendable-output function. Because those layers are public, the choice of generator does not affect the measured circuit structure, and hence not the cost comparison. It does mean the reference security claims do not transfer automatically: that would require the generated constants to match the intended distribution. The two are therefore evaluated as structurally faithful instances rather than byte-exact reference implementations. Since the homomorphic cost depends on circuit structure and not on the exact public constants, the comparison is unaffected.

Another generalisation limitation is that the comparison is tied to this TPSI construction. The workload is deliberately bit-oriented and multi-party, which makes it a useful stress test for Boolean FHE and threshold circuits. Other workloads may favour different backends. The framework design makes those workloads easier to add, but they are not measured here.

The comparison breadth is also limited. The high-N sweeps extend only two pipelines, Kreyvium/-Boolean and Pasta/BFV. The other four are measured at $N = 3$, $T = 2$, so the finding that no single pipeline wins rests on that point for them, with their high-N behaviour projected from the per-bit cost model rather than measured directly. Each (N, T, C, S) configuration is also a single seeded dataset realisation rather than a seed-averaged result, so the reported false-positive counts carry the sampling variation of one draw.

Finally, the system is evaluated under the single-key, honest-but-curious model from Chapter 4. This is a trust-model scope rather than a cost-result limitation. A threshold-decryption deployment would strengthen the trust model by removing the single decryption holder, but it would not change the main server-side cost comparison measured here.

7.3. Future Work

One important direction is to replace the bitwise SOP threshold with a more arithmetic threshold representation. The current SOP circuit grows with $\binom{N}{T}$ and becomes infeasible on per-gate-bootstrapped backends at high N. An arithmetic sum-and-compare circuit could reduce the dependence on the combinatorial SOP, especially on backends that handle integer arithmetic or packed SIMD more naturally. This would directly address the high-N bottleneck observed in the Kreyvium sweep.

The implementation could also be improved through backend-specific HE optimisations. The implementation intentionally keeps the comparison clean and portable across pipelines, but production implementations could use hoisting, better key-switch amortisation, packed rotations, parallel scheduling, or circuit layouts optimised by hand. These would not remove the fundamental trade-offs, but they could improve absolute runtimes.

Another useful extension would be to broaden the backend set. In particular, a GF(2)-SIMD backend would make it possible to evaluate XOR-heavy binary designs such as Rasta variants in a more natural packed representation. This is also where Dasta, a fixed-layer Rasta variant, would be a useful controlled reference point [46]. It is not run here because the selected BFV implementation is prime-field based and does not provide GF(2) batching.

A fourth direction is to study alternative FHE-friendly ciphers under the same workload. The results show that FHE-friendly is not a single category. Kreyvium and Grain both improve over AES, but Kreyvium is substantially better in this workload. Rasta/shortint is technically the right backend match for Rasta among the implemented options, but remains uncompetitive. A broader cipher suite would make the Pareto outcome more complete. The small-integer TFHE-native designs excluded in Chapter 5, such as Transistor, FRAST, and Elisabeth, belong here too, under a different premise: their high-throughput small-integer keystream pays off only when the downstream computation is itself small-integer or lookup-table shaped, so they become priority candidates once the workload enables more than just single-bit operations.

Beyond TPSI, Transieve could be applied to richer workloads. TPSI is useful because it is bit-oriented, multi-party, and communication-heavy, but Transieve is designed to separate the workload from the backend and transciphering layers. Secure voting, private telemetry aggregation, or integer-valued risk scoring would test whether the same pipeline conclusions hold outside TPSI.

The security model could also be strengthened through authenticated transciphering. Transciphering protects confidentiality but not integrity: under the honest-but-curious model of Section 4.2 the server is trusted to evaluate the agreed circuit, but a malicious server could tamper with the symmetric input or return an incorrect result without detection. Binding the homomorphic result to the claimed computation and its inputs would extend the guarantees beyond the semi-honest setting. It is a security extension and has an impact on the cost, through homomorphic verification, but it sits outside the measurement aim of this thesis.

A further direction is to lift the BFV depth ceiling by bootstrapping, since the BFV security ceiling is the single most prominent BFV limitation. The Pasta/BFV pipeline runs leveled, so its securable party count is capped by the modulus budget (Section 5.4.3). A bootstrapped BFV backend would refresh noise mid-circuit and remove that bound, at a per-bootstrap cost this thesis does not measure. This would test whether the BFV transciphering path can be pushed past the $N \approx 4$ ceiling while holding 128-bit security, and needs a backend providing BFV bootstrapping such as OpenFHE or SEAL. This would allow for Rasta to be measured on a more fitting backend.

Transieve could also evolve from an evaluation framework into a design aid for FHE-friendly ciphers. Because it reports a candidate cipher's measured per-bit cost, alongside its specification-derived bootstrap count and multiplicative depth, across three backends and against a real workload, it provides efficiency feedback based on the backends. With further development it could serve as the evaluation metric for automated cipher parameter tuning, searching over choices such as round count, S-box degree, and linear-layer density to minimise homomorphic cost. Security analysis would remain outside its scope and be supplied separately, but this would extend Transieve from a tool that compares existing ciphers into one that helps design new ones.

Finally, threshold FHE could be added to strengthen the trust model. In the current implementation, the

result is decrypted under a single key holder. A threshold variant would distribute decryption authority across parties. This is important for deployment, but secondary to the main aim of this thesis: the measured transciphering overhead and bandwidth trade-offs would largely transfer, because they are server-side evaluation costs.

7.4. Conclusion

This thesis asked how transciphering pipelines compare under a fair, end-to-end comparison on one workload, threshold private set intersection, at a common 128-bit security level. The answer is that transciphering is effective, but not universally optimal. It removes the upload expansion that makes direct FHE unattractive for large inputs, but it introduces a homomorphic symmetric-decryption cost whose size depends on the cipher/backend match.

RQ1 asked what it costs to homomorphically evaluate each backend's best-fit cipher. The measured per-bit costs span almost four orders of magnitude. Pasta/BFV is the fastest at about 4.1 ms/bit. Kreyvium/Boolean is the best Boolean pipeline at about 313 ms/bit. Grain/Boolean is slower, around 1.1 s/bit. AES costs about 4–5 s/bit on the Boolean and shortint backends. Rasta/shortint is the slowest measured pipeline, with around 18.5 s/bit.

RQ2 asked how upload saving scales with data size. The Boolean pipelines show the strongest result: at $S = 10^6$, direct Boolean FHE uploads 87.30 GiB per party, while transciphering uploads 3.83 MiB. The fixed encrypted-key cost dominates tiny inputs, but is amortised away at large inputs. Pasta/BFV also saves bandwidth over direct BFV, but only by about $16\times$ at $S = 10^6$, because direct BFV is already compact and Pasta uses prime-field words.

RQ3 asked whether there is a single best pipeline. There is not. The comparison is a Pareto outcome. Kreyvium/Boolean is the minimal bandwidth point. Pasta/BFV is the minimal compute point when the party count is low enough to fit the leveled BFV depth budget and the server can afford the memory. The other pipelines are dominated or negative results, but they are important because they demonstrate and explain why cipher/backend matching matters.

RQ4 asked how the protocol scales. The Bloom-filter threshold protocol has perfect recall in the measured sweeps, but false positives depend strongly on the threshold ratio. The SOP circuit grows combinatorially with N and T, peaking near half thresholds. At high party counts, this threshold circuit becomes the dominant cost and eventually overwhelms the benefits of improving the transciphering cipher.

RQ5 asked how much algebra/backend matching matters. It matters decisively. AES is expensive under FHE despite being efficient on normal processors. Kreyvium matches Boolean TFHE much better than AES or Grain. Pasta matches BFV's prime-field SIMD model and is therefore the throughput winner. Rasta/shortint shows that choosing a better backend for a cipher is not sufficient if the resulting circuit still contains too much work that the backend does not make cheap.

RQ6 asked how Bloom-filter accuracy trades against bandwidth. Larger filters reduce the observed false-positive rate but increase the payload. The protocol-level FPR can remain much higher than the per-filter target because the threshold OR-of-ANDs amplifies false positives. Threshold selection is therefore as important as Bloom-filter sizing: the upper-middle threshold region gives much better precision, but also coincides with the largest SOP circuits.

The final conclusion is that transciphering should be evaluated as a complete pipeline. A cipher, an FHE backend, and a workload cannot be ranked independently. The best choice depends on what the deployment is short of: bandwidth, compute, memory, or feasible multiplicative depth. Transieve makes this comparison explicit for threshold PSI and provides a reusable way to make the same comparison for future workloads and future transciphering candidates. Transciphering itself is a young technique and its landscape is still evolving. New FHE-friendly ciphers such as Transistor [49] and FRAST [50] appeared recently, and the surrounding methods are still being systematised [22, 24]. Each new proposal raises the question this thesis asked, namely whether the pipeline it enables pays off for a given workload. Transieve was built so that answering that question becomes easier: a new candidate slots in behind an existing interface and is measured under the same workload, security level, and metrics as everything before it.

References

[1] Wavestone. *Data and AI Leadership Executive Survey 2024*. Survey of data and AI leaders at large enterprises. 2024. URL: https://www.wavestone.com/wp-content/uploads/2024/11/data-ai-executive-leadership-survey-2024.pdf (visited on 06/15/2026).

[2] U.S. Senate Permanent Subcommittee on Investigations. *Protecting Consumer Data in the Cloud*. Tech. rep. Committee print; case study: the 2019 Capital One data breach. United States Senate Committee on Homeland Security and Governmental Affairs, Nov. 2022.

[3] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 (General Data Protection Regulation)*. Official Journal of the European Union, L 119. 2016.

[4] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Tech. rep. Special Publication 800-145. National Institute of Standards and Technology, 2011.

[5] Gartner, Inc. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Total $723 Billion in 2025*. Press release. 2024. URL: https://www.gartner.com/en/newsroom/press-releases/2024-11-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-total-723-billion-dollars-in-2025 (visited on 06/08/2026).

[6] Michael Armbrust et al. “A View of Cloud Computing”. In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[7] Subashini Subashini and Veeraruna Kavitha. “A Survey on Security Issues in Service Delivery Models of Cloud Computing”. In: *Journal of Network and Computer Applications* 34.1 (2011), pp. 1–11.

[8] Thomas Ristenpart et al. “Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. 2009, pp. 199–212.

[9] Nir Ohfeld and Sagi Tzadik. *ChaosDB: How We Hacked Thousands of Azure Customers’ Databases*. Wiz Research; cross-tenant vulnerability in Azure Cosmos DB. 2021. URL: https://www.wiz.io/blog/chaosdb-how-we-hacked-thousands-of-azure-customers-databases (visited on 06/14/2026).

[10] Abbas Acar et al. “A Survey on Homomorphic Encryption Schemes: Theory and Implementation”. In: *ACM Computing Surveys* 51.4 (2018), pp. 1–35.

[11] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*. 2009, pp. 169–178.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Paper 2011/277. 2011. URL: https://eprint.iacr.org/2011/277.

[13] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. 2012. URL: https://eprint.iacr.org/2012/144.

[14] Jung Hee Cheon et al. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Paper 2016/421. 2016. URL: https://eprint.iacr.org/2016/421.

[15] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption over the Torus*. Cryptology ePrint Archive, Paper 2018/421. 2018. URL: https://eprint.iacr.org/2018/421.

[16] Apple Machine Learning Research. *Combining Machine Learning and Homomorphic Encryption in the Apple Ecosystem*. 2024. URL: https://machinelearning.apple.com/research/homomorphic-encryption.

[17] Microsoft Research. *Password Monitor: Safeguarding Passwords in Microsoft Edge*. 2021. URL: https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/.

[18] Anne Canteaut et al. *Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression*. Cryptology ePrint Archive, Paper 2015/113. 2015. URL: `https://eprint.iacr.org/2015/113`.

[19] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. "Can Homomorphic Encryption be Practical?" In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW)*. 2011, pp. 113–124.

[20] Martin R. Albrecht et al. "Ciphers for MPC and FHE". In: *Advances in Cryptology – EUROCRYPT 2015*. 2015, pp. 430–454.

[21] Martin Albrecht et al. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*. Cryptology ePrint Archive, Paper 2016/492. 2016. URL: `https://eprint.iacr.org/2016/492`.

[22] Hossein Abdinasibfar, Camille Nuoskala, and Antonis Michalas. *The HHE Land: Exploring the Landscape of Hybrid Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2025/071. Published in ACM Computing Surveys 58(12), 2026. 2025. URL: `https://eprint.iacr.org/2025/071`.

[23] Martin Albrecht et al. *Homomorphic Encryption Standard*. HomomorphicEncryption.org. 2018. URL: `https://homomorphicencryption.org/standard/`.

[24] Chao Niu et al. "SoK: FHE-Friendly Symmetric Ciphers and Transciphering". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.3 (2025), pp. 583–613. DOI: `10.46586/tches.v2025.i3.583-613`.

[25] Zama. *TFHE-rs: A Pure-Rust Implementation of TFHE*. Version 0.8. 2024. URL: `https://github.com/zama-ai/tfhe-rs` (visited on 06/07/2025).

[26] Tancrède Lepoint. *fhe.rs: Fully Homomorphic Encryption in Rust*. Crate `fhe` v0.1.x. 2022. URL: `https://github.com/tlepoint/fhe.rs` (visited on 06/07/2025).

[27] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[28] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology – EUROCRYPT 1999*. 1999, pp. 223–238.

[29] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Journal of the ACM* 56.6 (2009), 34:1–34:40.

[30] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Journal of the ACM* 60.6 (2013), 43:1–43:35.

[31] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel P. Smart. *Trivial Transciphering with Trivium and TFHE*. Cryptology ePrint Archive, Paper 2023/980. 2023. URL: `https://eprint.iacr.org/2023/980`.

[32] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. Tech. rep. FIPS PUB 197. NIST, 2001.

[33] Christophe De Cannière. "Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles". In: *Information Security (ISC)*. 2006, pp. 171–186.

[34] Martin Ågren et al. "Grain-128a: A New Version of Grain-128 with Optional Authentication". In: *International Journal of Wireless and Mobile Computing* 5.1 (2011), pp. 48–59. DOI: `10.1504/IJWMC.2011.044106`.

[35] Christoph Dobraunig et al. *Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit*. Cryptology ePrint Archive, Paper 2018/181. 2018. URL: `https://eprint.iacr.org/2018/181`.

[36] Claude E. Shannon. "Communication Theory of Secrecy Systems". In: *Bell System Technical Journal* 28.4 (1949), pp. 656–715.

[37] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *Journal of Cryptology* 4.1 (1991), pp. 3–72.

[38] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher". In: *Advances in Cryptology – EUROCRYPT 1993*. 1993, pp. 386–397.

[39] Nicolas T. Courtois and Josef Pieprzyk. “Cryptanalysis of Block Ciphers with Overdefined Systems of Equations”. In: *Advances in Cryptology – ASIACRYPT 2002*. 2002, pp. 267–287.

[40] Jihoon Cho et al. *Transciphering Framework for Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2020/1335. 2020. URL: https://eprint.iacr.org/2020/1335.

[41] Jincheol Ha et al. “Rubato: Noisy Ciphers for Approximate Homomorphic Encryption”. In: *Advances in Cryptology – EUROCRYPT 2022*. 2022, pp. 581–610.

[42] Jincheol Ha et al. “Masta: An HE-Friendly Cipher Using Modular Arithmetic”. In: *IEEE Access* 8 (2020), pp. 194741–194751.

[43] Christoph Dobraunig et al. *Pasta: A Case for Hybrid Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2021/731. 2021. URL: https://eprint.iacr.org/2021/731.

[44] Khoa Nguyen et al. *A Pervasive, Efficient and Private Future: Realizing Privacy-Preserving Machine Learning Through Hybrid Homomorphic Encryption*. arXiv preprint arXiv:2409.06422. 2024. URL: https://arxiv.org/abs/2409.06422.

[45] Kelong Cong et al. “SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022, pp. 563–577.

[46] Phil Hebborn and Gregor Leander. “Dasta – Alternative Linear Layer for Rasta”. In: *IACR Transactions on Symmetric Cryptology* 2020.3 (2020), pp. 46–86.

[47] Orel Cosseron et al. *Towards Case-Optimized Hybrid Homomorphic Encryption – Featuring the Elisabeth Stream Cipher*. Cryptology ePrint Archive, Paper 2022/180. 2022. URL: https://eprint.iacr.org/2022/180.

[48] Pierrick Méaux et al. *Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts*. Cryptology ePrint Archive, Paper 2016/254. 2016. URL: https://eprint.iacr.org/2016/254.

[49] Jules Baudrin et al. *Transistor: A TFHE-friendly Stream Cipher*. Cryptology ePrint Archive, Paper 2025/282. 2025. URL: https://eprint.iacr.org/2025/282.

[50] Mingyu Cho et al. *FRAST: TFHE-friendly Cipher Based on Random S-boxes*. Cryptology ePrint Archive, Paper 2024/745. 2024. URL: https://eprint.iacr.org/2024/745.

[51] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. “Efficient Private Matching and Set Intersection”. In: *Advances in Cryptology – EUROCRYPT 2004*. 2004, pp. 1–19.

[52] Lea Kissner and Dawn Song. “Privacy-Preserving Set Operations”. In: *Advances in Cryptology – CRYPTO 2005*. 2005, pp. 241–257.

[53] Hao Chen, Kim Laine, and Peter Rindal. *Fast Private Set Intersection from Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2017/299. 2017. URL: https://eprint.iacr.org/2017/299.

[54] Kelong Cong et al. *Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication*. Cryptology ePrint Archive, Paper 2021/1116. 2021. URL: https://eprint.iacr.org/2021/1116.

[55] Mrityunjaya Palanimurugan Vasanthakumari. “Privacy Preserving Collaborative Attack Campaign Detection”. MSc thesis. Delft University of Technology, 2026. URL: https://resolver.tudelft.nl/uuid:641f661e-47d5-4be9-862e-31dd0b3e5f78.

[56] Chelsea Guan, Jorrit van Assen, and Zekeriya Erkin. “Collective Threshold Multiparty Private Set Intersection Protocols for Cyber Threat Intelligence”. In: *2024 IEEE International Workshop on Information Forensics and Security (WIFS)*. Rome, Italy: IEEE, 2024.

[57] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets”. In: *27th Annual Symposium on Foundations of Computer Science (FOCS)*. 1986, pp. 162–167.

[58] Cynthia Dwork. “Differential Privacy”. In: *Automata, Languages and Programming (ICALP)*. 2006, pp. 1–12.

[59] Fukang Liu et al. *Algebraic Attacks on Rasta and Dasta Using Low-Degree Equations*. Cryptology ePrint Archive, Paper 2021/474. 2021. URL: https://eprint.iacr.org/2021/474.

[60] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[61] Joan Boyar and René Peralta. *A New Combinational Logic Minimization Technique with Applications to Cryptology*. Cryptology ePrint Archive, Paper 2009/191. 2009. URL: https://eprint.iacr.org/2009/191.

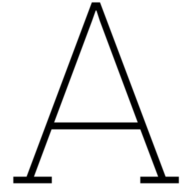
[62] Shai Halevi and Victor Shoup. *Bootstrapping for HElib*. Cryptology ePrint Archive, Paper 2014/873. 2014. URL: https://eprint.iacr.org/2014/873.

[63] Jelle Vos, Mauro Conti, and Zekeriya Erkin. *Depth-Aware Arithmetization of Common Primitives in Prime Fields*. Cryptology ePrint Archive, Paper 2024/1200. 2024. URL: https://eprint.iacr.org/2024/1200.

[64] Shai Halevi and Victor Shoup. *Faster Homomorphic Linear Transformations in HElib*. Cryptology ePrint Archive, Paper 2018/244. 2018. URL: https://eprint.iacr.org/2018/244.

[65] Adam Kirsch and Michael Mitzenmacher. “Less Hashing, Same Performance: Building a Better Bloom Filter”. In: *Algorithms – ESA 2006*. 2006, pp. 456–467.

[66] Nigel P. Smart and Frederik Vercauteren. “Fully Homomorphic SIMD Operations”. In: *Designs, Codes and Cryptography* 71.1 (2014), pp. 57–81.



Implementation Parameters and Reproducibility

The implementation is available at `https://github.com/RRas/Transieve`.

This appendix records the concrete parameters behind the measurements of Chapter 6, so that the comparison can be reproduced. All three backends target the same $\lambda = 128$-bit security level (Section 6.1); within that constraint each backend uses the parameter set its library provides.

A.1. FHE backend parameters

Table A.1: FHE parameters per backend. The TFHE backends use the `tfhe-rs` library parameter sets; the BFV backend uses `fhe.rs` in leveled mode.

Backend	Library	Parameter set	Plaintext
TFHE-Boolean	`tfhe-rs`	default Boolean parameters (`gen_keys()`), one bit per ciphertext, gate bootstrapping	1 bit
TFHE-shortint	`tfhe-rs`	`PARAM_MESSAGE_2_CARRY_2_KS_PBS` (2 message $+ 2$ carry bits)	small integer
BFV	`fhe.rs`	leveled; ring degree chosen per run as the smallest 128-bit-secure power of two ≥ 8192 whose modulus chain fits the circuit depth, capped at 32768 (max ≈ 881 bits, HES [23])	$\mathbb{F}_p$, $p = 65537$

The BFV ring degree is not fixed: `secure_degree_128` selects the smallest secure power-of-two degree whose modulus chain covers the required multiplicative depth. Because the plaintext modulus $p = 65537$ admits full SIMD batching only for degrees up to 32768, a circuit whose chain exceeds the budget at that degree cannot be evaluated at 128-bit security; this is the security ceiling of Section 5.4.3.

A.2. Sampled-prefix caps

The FHE timings are measured on a sampled prefix of the Bloom filter and extrapolated (Section 6.2). The per-bit cost is bits-independent for these circuits, so the cap affects feasibility, not the reported per-bit figure.

A.3. Hardware and determinism

All runs were executed as DelftBlue HPC jobs on a fixed 16-core allocation, compiled with native CPU optimisations (`target-cpu=native`) and a fixed Rayon thread count pinned to the allocated cores.

Table A.2: Sampled-prefix size per pipeline. In the Pasta-3/BFV high-N sweep the ring degree, and with it the batch, grows with the circuit depth.

Pipeline	Set-size sweep	High-N sweep
Kreyvium / Boolean	8192 bits	64 bits
Grain-128a / Boolean	512 bits (`sb512` rerun)	—
AES-128 / Boolean	512 bits	—
AES-128 / shortint	512 bits	—
Rasta-525-5 / shortint	512 bits	—
Pasta-3 / BFV	SIMD batch (≥ 8192, up to 32768)	8192–32768 bits

Datasets and keys are generated from a deterministic seeded RNG, so repeated runs are byte-identical and the plaintext evaluation is an exact correctness oracle for every FHE path. Wall-clock times are single-shot per sweep point; the hardware-independent counts (PBS per output bit, multiplicative depth; Table 5.2) are the portable axis.

B
The Transieve Interfaces

The decoupling claimed in Chapter 5 rests on three narrow interfaces, one per layer. This appendix lists them as they appear in the implementation, to make the "swap one component, keep the rest" property concrete. A new workload, cipher, or backend is added by implementing the corresponding trait; nothing else in the pipeline changes.

B.1. Workload layer

A use case implements `Application` to build a `Workload`: the N bit-operands, the Boolean circuit over them, the plaintext oracle, and the decode. Threshold PSI and the voting demonstration are the two implementations.

```
1  // A built, encoded workload: N bit-operands + the SOP over them,
2  // plus the use-case oracle and decode. Consumed by the pipeline.
3  pub trait Workload {
4      fn operands(&self) -> &[Vec<bool>]; // N equal-length bit-vectors
5      fn circuit(&self) -> &ThresholdCircuit; // slot-wise boolean SOP
6      fn plaintext_result(&self) -> &[bool]; // correctness oracle
7      fn recovered_count(&self, result_bits: &[bool]) -> usize; // decode
8      fn accuracy(&self) -> Accuracy; // ground truth + pool
9      fn encoding(&self) -> EncodingInfo; // operand length, #hashes
10     fn run_tag(&self) -> String;
11     fn summary(&self) -> String;
12 }
13
14 // A use case: builds a Workload from the generic N / threshold / size knobs.
15 pub trait Application {
16     fn name(&self) -> &'static str;
17     fn slug(&self) -> &'static str;
18     fn build(&self, cfg: &SampleConfig, root: &Path) -> Result<Box<dyn Workload>>;
19 }
```

B.2. Transciphering layer

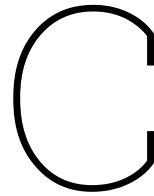
Every symmetric cipher exposes the same operation: recover FHE ciphertexts of the data from a symmetric ciphertext and an FHE-encrypted key. The trait also carries the spec-derived, hardware-independent cost constants of Table 5.2. The Boolean-backend version is shown; the shortint and BFV backends expose the analogous trait over their own ciphertext type.

```
1  // A symmetric cipher's decryption circuit, evaluated under FHE (Boolean).
2  pub trait TranscipherDecrypt {
3      const BLOCK_BITS: usize;
4      const KEY_BITS: usize;
5      const NAME: &'static str;
6      const PBS_PER_OUTPUT_BIT: f64; // spec-derived cost (Table 5.x)
7      const MULTIPLICATIVE_DEPTH: usize; // AND-depth of the circuit
8      const FIXED_INIT_PBS: usize; // one-off warm-up / key schedule
```

```
9
10     // Recover FHE bits from the encrypted key and one symmetric ct block.
11     fn decrypt_block(sk: &ServerKey,
12                      enc_key: &[EncBit],
13                      cipher_block: &[bool]) -> Vec<EncBit>;
14 }
```

B.3. FHE backend layer

The backend supplies the primitive operations the layers above are written against: encryption and decryption, the bit/word gates (`AND`, `OR`, `XOR` on the TFHE backends; add and multiply with SIMD on BFV), and key generation. Each backend is a separate module (`fhe.rs`, `fhe_shortint.rs`, `fhe_bfv.rs`) implementing these primitives, so the same workload and cipher run unchanged on any of the three.



Supplementary Pipeline Figures

This appendix collects the per-pipeline figures that back the consolidated comparison of Chapter 6. For each pipeline it shows the per-stage latency breakdown (left) and the wall-clock latency across the set-size sweep (right), at $N = 3$, $T = 2$, $C = 10$. The accuracy and circuit-size plots are not repeated per pipeline: accuracy is a property of the workload and the threshold circuit is identical across ciphers, so both are already given once in Chapter 6.

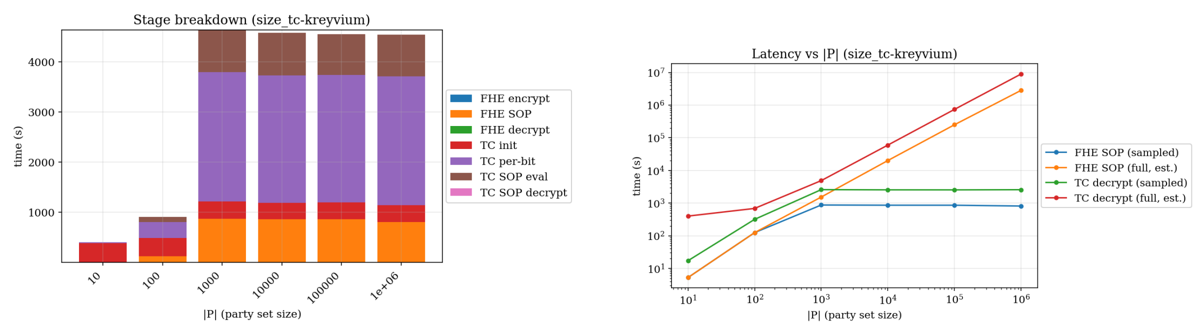


Figure C.1: Kreyvium/Boolean: stage breakdown (left) and latency (right) across the set-size sweep.

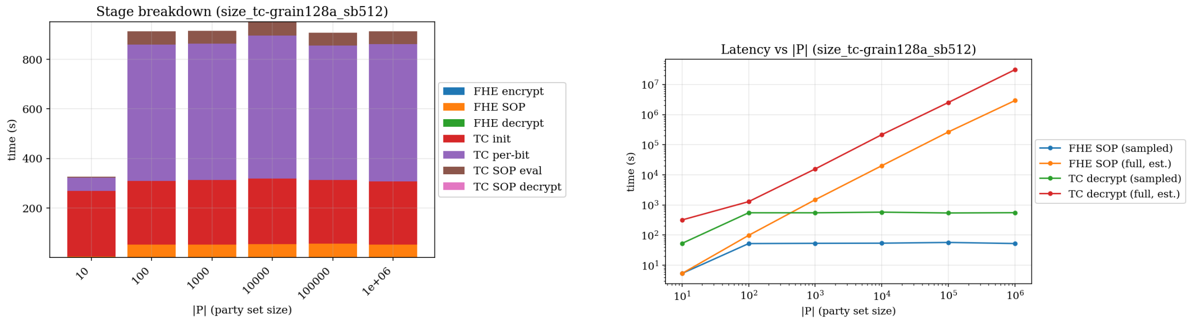


Figure C.2: Grain-128a/Boolean (512-bit rerun): stage breakdown (left) and latency (right).

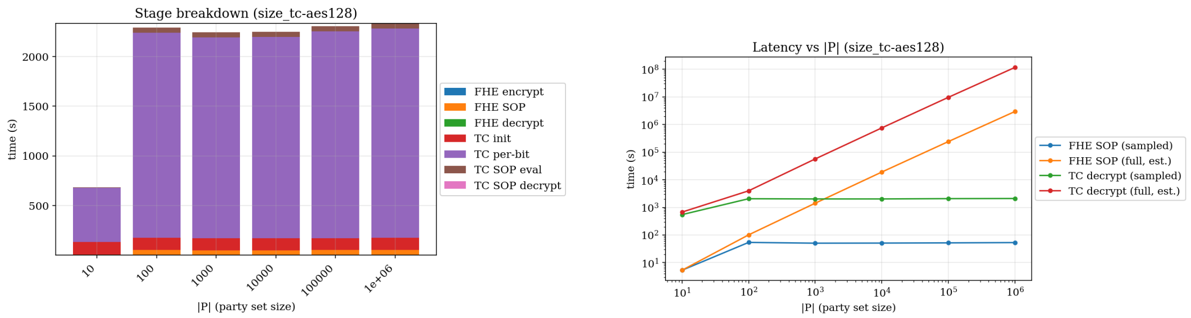


Figure C.3: AES-128/Boolean: stage breakdown (left) and latency (right).

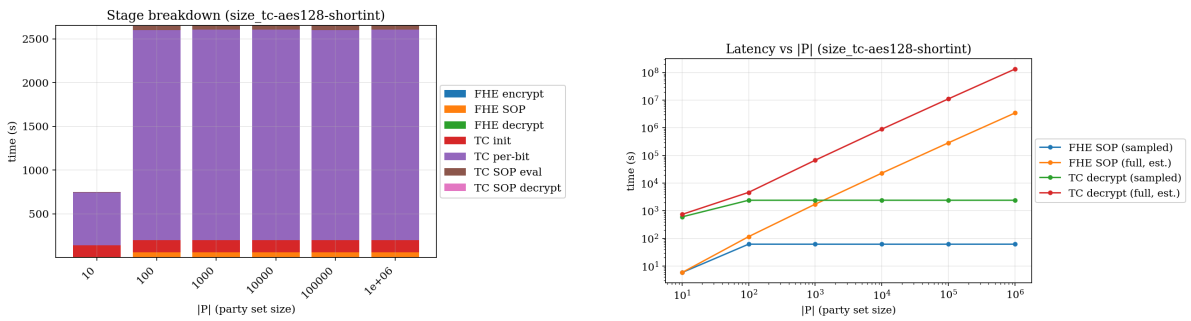


Figure C.4: AES-128/shortint: stage breakdown (left) and latency (right).

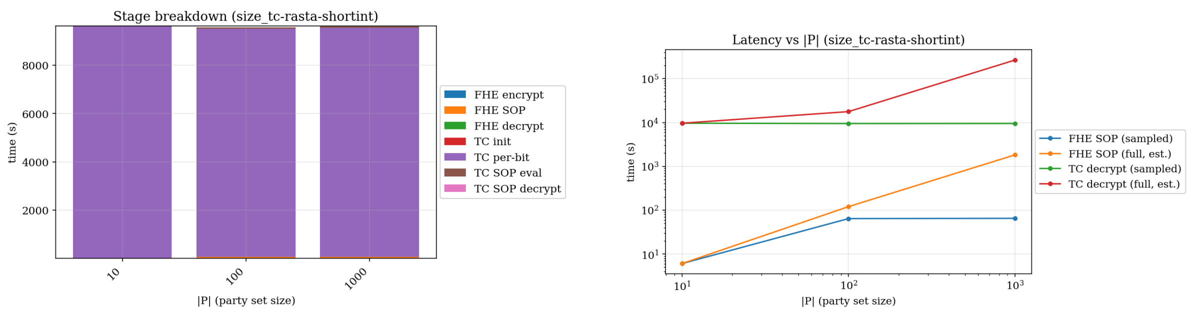


Figure C.5: Rasta-525-5/shortint: stage breakdown (left) and latency (right).

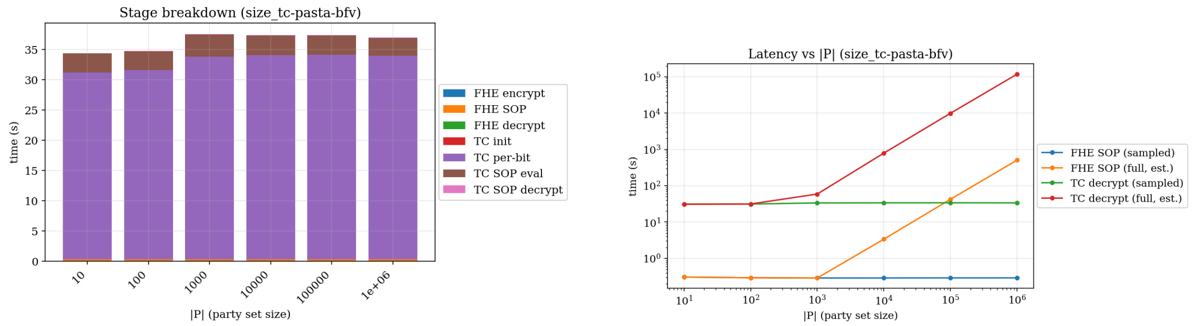


Figure C.6: Pasta-3/BFV: stage breakdown (left) and latency (right).

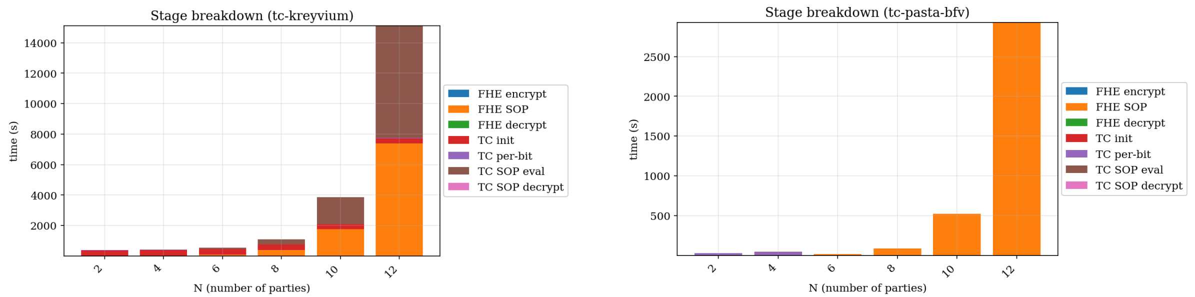


Figure C.7: High-N stage breakdowns: Kreyvium/Boolean (left) and Pasta/BFV (right) over the party sweep, showing the SOP overtaking the cipher-decrypt stage as N grows.

D

Voting Workload Demonstration Results

This appendix shows the results of the voting app demo, transciphering the workload with Kreyvium-128 on the Boolean TFHE backend. Unlike the measurements of Chapter 6, this demonstration was run on a local machine rather than the DelftBlue configuration, so its absolute timings (about 119 ms per Kreyvium keystream bit, against the 313 ms per bit of Table 6.3) are not comparable with the benchmark campaign. The run demonstrates the workload seam and functional correctness, not performance.

Listing D.1: Terminal output for N=3, T=2 simulation.

```
[RUN] vote_n3_q2_p100 | app=Secure threshold voting | Voting: 3 voters, quorum 2, 100
    proposals
[RUN] Circuit: 3 terms, 3 AND/bit, 2 OR/bit -> AB + AC + BC
[RUN] Plaintext recovered 48; exact answer 48
[RUN] Generating TFHE Boolean keys ...
[RUN] FHE baseline (boolean): evaluating 100 of 100 bits (100%)
[RUN] FHE sample matches plaintext prefix: PASS

[TRANSCIPHER boolean TPSI] Kreyvium-128 on 3 party filter(s), 100 bit(s) each
[TRANSCIPHER] party 1/3: homomorphic decrypt (100 bits) ...
  party 0: init=128.069s, per-bit total=11.926s (119.3 ms/bit)
[TRANSCIPHER] party 2/3: homomorphic decrypt (100 bits) ...
  party 1: init=127.489s, per-bit total=12.005s (120.1 ms/bit)
[TRANSCIPHER] party 3/3: homomorphic decrypt (100 bits) ...
  party 2: init=127.550s, per-bit total=11.877s (118.8 ms/bit)
[TRANSCIPHER] all-parties roundtrip: PASS
[TRANSCIPHER] evaluating Boolean SOP (3 terms, 100 bits) ...
[TRANSCIPHER] SOP eval=4.035s, decrypt=0.000s
[TRANSCIPHER] SOP match vs plaintext: PASS
[TRANSCIPHER] recovered 48 candidate(s) from Boolean TPSI

==============================================================================
Sample run
==============================================================================
run load bloom plain fhe_enc fhe_eval fhe_dec
------------------------------------------------------------------------------
vote_n3_q2_p100 0.001s n/a n/a 0.001s 3.947s 0.000s

------------------------------------------------------------------------------------------------
transcipher cipher backend init per-bit ms/bit bits thrpt b/s match
------------------------------------------------------------------------------------------------
vote_n3_q2_p100 Kreyvium-128 boolean 127.702s 11.936s 119.4 100 8.4 PASS
```

```
32
33 --------------------------------------------------------------------------------------------
34 cipher metadata cipher PBS/out_bit depth fixed_init_PBS
35 --------------------------------------------------------------------------------------------
36 vote_n3_q2_p100 Kreyvium-128 3.00 1 3456
37
38 --------------------------------------------------------------------------------------------
39 transcipher TPSI SOP parties sop_eval sop_dec sop_total sop_ok recovered
40 --------------------------------------------------------------------------------------------
41 vote_n3_q2_p100 3 4.035s 0.000s 4.035s PASS 48
42
43 ------------------------------------------------------------------------------
44 size / bandwidth plain fhe_direct transcipher exp_fhe exp_tc
45 ------------------------------------------------------------------------------
46 vote_n3_q2_p100 13 B 318.36 KiB 407.51 KiB 25076.9x 32099.5x
47   vote_n3_q2_p100 bandwidth savings (fhe_direct / transcipher) = 0.8x
48   vote_n3_q2_p100 transcipher cost = 119.36 ms/bit; full filter est. 139.6s
49   vote_n3_q2_p100 peak RSS = 182.69 MiB
50 ==============================================================================
```