Logic Synthesis Foundations for Efficient Secure Computation:

From Fully Homomorphic Encryption to Garbled Circuits

présenté le 31 Octobre 2025
à la Faculté des Sciences et Techniques de l'Ingénieur (STI)
Laboratoire des Systèmes Intégrés (LSI)
Programme Doctoral en Génie Électrique
École Polytechnique Fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Mingfei Yu

acceptée sur proposition du jury:

Prof. Mario Paolone, président du jury
Prof. Giovanni De Micheli, directeur de thèse
Prof. David Atienza, codirecteur de thèse
Prof. Paolo Ienne, rapporteur
Prof. Makoto Ikeda, rapporteur
Prof. Ingrid Verbauwhede, rapporteur

Lausanne, EPFL, 2025

Never put off until runtime what you can do at compile time.
— John Hennessy

To all the glorious entanglements that found their way to me.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my PhD advisor, Professor Giovanni De Micheli, and my co-advisor, Professor David Atienza. Nanni, in every sense, is a remarkably wise and generous mentor. Over the past four years, there have been countless moments when I was enlightened by his insight, both within research and far beyond. Despite my early-stage immaturity as a researcher, Nanni entrusted me with the freedom to pursue the problems that fascinated me the most. This freedom, paired with his constant and unconditional support, created an ideal environment for my intellectual growth. I have felt this support ever since receiving his very first reply during my master's studies, and it has never ceased. I could never thank him enough for that. To David, I owe my sincere appreciation as well. Although our interactions were less frequent, they were always impactful. I was consistently inspired by his boundless energy and enthusiasm, even during an incredibly demanding schedule. His support, too, has been timely and wholehearted. I am confident that, when I have the opportunity to mentor the next generation of researchers, my philosophy will carry the influence of both Nanni and David.

I extend my gratitude to the members of my dissertation jury: Professor Mario Paolone, who served as president, and Professors Paolo Ienne, Makoto Ikeda, and Ingrid Verbauwhede, who generously acted as reviewers. I sincerely appreciate their time and effort in reviewing my dissertation and participating in the intense but unforgettable oral defense. I still vividly remember how each expert's questions reflected their unique research perspectives and personalities, making the discussion both challenging and intellectually rewarding. I remain amazed by the new ideas that emerged during that exchange and am deeply thankful for the opportunity to engage with them.

I would also like to thank my master's advisor, Professor Masahiro Fujita. Through his example, Fujita-sensei has continually reminded me to stay honest with myself in conducting research, remaining open-minded while having the courage to pursue questions that truly intrigue me.

I am immensely grateful to the LSIers — my colleagues at the Integrated Systems Laboratory. To those who overlapped with my four years at LSI: Dr. Mathias Soeken, Dr. Rassul Bairamkulov, Professor Chang Meng, Dr. Rehab Massoud, Dr. Bruno Schmidt, Dr. Fereshte Mozafari, Dr. Siang-Yun (Sonia) Lee, Dr. Alessandro Tempia Calvino, Dr. Dewmini Sudara Marakkalage, Andrea Costamagna, Junrui Chen, Hanyu Wang, and Julien de Castelnau, it was a great privilege to share this time with you in such an extraordinary research group.

Our interactions extended far beyond research: through them, I gradually learned that excellence, humility, and kindness can coexist. Immersed in this supportive and encouraging environment, I slowly gained the strength to acknowledge my limitations, embrace failure, and find the courage to begin again. My heartfelt thanks also go to our wonderful secretary, Ms. Chantal Demont. Chantal's tireless efforts, whether handling countless administrative tasks or organizing team-building events, ensured that our lab remained lively and focused. Her behind-the-scenes work made our research lives significantly smoother, and I am deeply appreciative of her dedication. Lastly, I would like to acknowledge the former LSI members whom I had the pleasure of meeting in various corners of the world: Professor Pierre-Emmanuel Gaillardon, Dr. Luca Amarù, Professor Zhufei Chu, Professor Cunxi Yu, Dr. Xifan Tang, Dr. Heinz Riener, Dr. Winston Haaswijik, Dr. Eleonora Testa, and Dr. Giulia Meuli. Thank you for your kindness, encouragement, and insightful guidance — you have helped me more than you know.

I deeply believe that meaningful breakthroughs — especially in interdisciplinary research areas such as design automation for cryptography — are rarely achieved in isolation. I would therefore like to express my sincere gratitude to my collaborators. My first project on graph-level optimization for leveled homomorphic encryption acceleration began to take shape thanks to fruitful discussions with Dr. Christian Mouchet, whose insights helped lay its foundation. Later, in collaboration with Dr. Sergiu Carpov and Dr. Gabrielle De Micheli, I found not only exceptional technical partners, but also generous and reliable colleagues who patiently corrected and enriched my understanding of modern FHE constructions. I would also like to thank the Computer Security Group at CWI, Amsterdam, led by Professor Marten van Dijk and Dr. Chenglu Jin. During my research internship, I was warmly welcomed into their group — a gesture I remain deeply grateful for.

I dedicate the next paragraph to all my mentors and friends. I have chosen not to name anyone specifically here, because I know I could never list all those who have profoundly shaped me. Your influence on my life and thinking has often come in subtle, hybrid ways. Many professors, researchers, and PhD students I encountered at academic conferences inspired me not only through their work but also through the life philosophies they embodied. At the same time, many of my closest friends work in entirely different areas, some not even in academia, but I find that my passion for certain research topics or philosophical ideas often echoes the spirit of our conversations. Everything is connected, and I feel truly fortunate to have been surrounded by people and moments that shaped who I am.

To my family: thank you for being the unwavering support that has anchored me through the highs and lows. You have absorbed my stress, anxiety, and doubts with patience and love. I am especially grateful to my grandparents and parents for always having my back — mentally, emotionally, and even financially — gently offering their guidance while respecting my choices. I feel deeply blessed to have such a family. I also want to express my love and gratitude to my wife, Yiru. After being together for seven years, we got married this year, which is a milestone that, to me, surpasses the PhD in significance and meaning. This journey has not been easy: the inherent uncertainty of research, combined with the complexity of life's choices, often threw us into overwhelming situations. But in navigating them together, we became each

other's constants — our "fixed parameters" in the grand decision-making problem of how to live a meaningful life. For that, I am endlessly thankful.
This world is changing rapidly, often provoking uncertainty and unease. The rise of large language models, for instance, is reshaping how we interact with knowledge and conduct research. And the world's darkness remains ever-present: the war between Russia and Ukraine broke out just before I began my PhD and still continues today, with many more conflicts arising in other corners of the world. Yet the more I write this acknowledgment, the more clearly I see how blessed I have been throughout this journey. This is the message I hope to leave here: you have all helped me resist being overwhelmed by negativity and reminded me to keep looking forward. With these words, I finally come to terms with the fact that this PhD chapter is drawing to a close — how luxurious it has been! But your influence will not end here; it will remain with me as a lifelong gift, equipping me for whatever comes next.

EPFL, November 16, 2025 Mingfei Yu

Abstract

As digital data become central to domains, ranging from personalized medicine and financial services to scientific research and national security, preserving its confidentiality during computation has emerged as a foundational challenge. Recent breaches illustrate the severe consequences of compromised information: privacy violations, identity theft, and loss of public trust. Traditional encryption protects data at rest and in transit, but leaves a critical gap during *computation*, when sensitive inputs are typically exposed. Secure computation techniques, notably *fully homomorphic encryption* (FHE) and *garbled circuits* (GC), close this gap by enabling meaningful computation over encrypted or distributed data without revealing the underlying inputs.

Despite remarkable progress, these techniques still suffer from high computational and communication overheads, limiting their practical deployment. This thesis targets a specific, but broadly applicable, scenario: the secure evaluation of Boolean functions using FHE and GC. While secure computing techniques are not inherently restricted to Boolean logic, focusing on Boolean circuits allows us to leverage decades of progress in *logic synthesis*, a mature subfield of *electronic design automation*. By casting secure computation as a domain-specific logic synthesis problem — one governed by cryptographic cost models rather than silicon constraints in the conventional context — we develop novel circuit optimization techniques that bridge classical logic design and modern cryptography.

We make contributions across five technical directions: (i) For leveled FHE schemes, we formalize the trade-off between *multiplicative depth* (MD) and *multiplicative complexity* (MC), and introduce synthesis frameworks that jointly reduce both, enabling tighter cryptographic parameters and faster homomorphic evaluation. (ii) For fast-bootstrapping FHE schemes, represented by the *torus FHE* (TFHE) scheme, under a fixed-plaintext-space function-evaluation strategy, we develop symmetry- and negacyclicity-aware mapping strategies, combined with a *multi-value programmable bootstrapping* (MV-PBS)–aware *lookup-table* (LUT) network synthesizer, to minimize PBS count. (iii) For TFHE-based, multi-plaintext-space function evaluation, we propose the first synthesis framework that strategically transitions between binary and large plaintext spaces, using XORs for linear logic and concentrating non-linear operations into compact LUTs. (iv) For GC, we introduce the *XOR–OneHot–inverter graph* (X1G), a new intermediate representation that improves ciphertext efficiency and unlocks dedicated optimization passes. (v) Finally, we introduce *joint multiplicative complexity* (JMC), a garbling-cost-aware cost model that accounts for shared ownership in GC-based secure multi-party

computation, and propose the first ownership-aware optimizer that reduces jointly evaluated non-linear gates.
Collectively, these results demonstrate two broad lessons: (1) *representation matters* — the choice of intermediate form directly impacts achievable savings and the scope of optimizations; and (2) *cost models must match cryptographic reality* — accurate modeling of performance bottlenecks is essential to obtain meaningful efficiency gains. Beyond these technical findings, the thesis argues for stronger standardization and cross-layer interfaces between cryptographic libraries, compilers, and hardware, to ensure that optimizations at one layer remain aligned with advances at others. By bridging logic synthesis with modern secure computation techniques, this work takes a step toward making cryptographic protocols not only theoretically powerful, but also practically usable at scale.

Key words: logic synthesis, cryptography-aware compiler design, secure computation, fully homomorphic encryption, garbled circuits

Résumé

À mesure que les données numériques deviennent essentielles dans des domaines allant de la médecine personnalisée et des services financiers à la recherche scientifique et à la sécurité nationale, la préservation de leur confidentialité durant le calcul s'impose comme un défi fondamental. Les incidents récents illustrent les conséquences graves d'une compromission de l'information : atteintes à la vie privée, vols d'identité et perte de confiance du public. Le chiffrement traditionnel protège les données au repos et en transit, mais laisse une faille critique durant la *phase de calcul*, lorsque les données sensibles sont généralement exposées. Les techniques de calcul sécurisé, notamment le *chiffrement entièrement homomorphe* (FHE) et les *circuits brouillés* (GC), comblent cette lacune en permettant d'effectuer des calculs utiles sur des données chiffrées ou distribuées sans révéler les entrées sous-jacentes.

Malgré des progrès remarquables, ces techniques souffrent encore de surcoûts computationnels et communicationnels élevés, limitant leur déploiement pratique. Cette thèse cible un scénario spécifique, mais largement applicable, l'évaluation sécurisée de fonctions booléennes à l'aide de FHE et de GC. Bien que les techniques de calcul sécurisé ne soient pas intrinsèquement limitées à la logique booléenne, se concentrer sur des circuits booléens permet de tirer parti de décennies de progrès en *synthèse logique*, un sous-domaine mature de la *conception assistée par ordinateur*. En reformulant le calcul sécurisé comme un problème de synthèse logique spécifique au domaine — régi par des modèles de coûts cryptographiques plutôt que par des contraintes liées au silicium dans le contexte conventionnel — nous développons de nouvelles techniques d'optimisation de circuits reliant la conception logique classique et la cryptographie moderne.

Nous apportons des contributions selon cinq directions techniques : (i) pour les schémas FHE à niveaux, nous formalisons le compromis entre *profondeur multiplicative* (MD) et *complexité multiplicative* (MC), et introduisons des cadres de synthèse réduisant conjointement les deux, permettant des paramètres cryptographiques plus serrés et une évaluation homomorphe plus rapide ; (ii) pour les schémas FHE à démarrage rapide, représentés par le schéma *Torus FHE* (TFHE), dans une stratégie d'évaluation à espace de clair fixé, nous développons des stratégies de mappage sensibles aux symétries et à la négacyclicité, combinées à un synthétiseur de réseaux de *tables de correspondance* (LUT) compatible avec le *programmable bootstrapping multi-valeurs* (MV-PBS), afin de minimiser le nombre de PBS ; (iii) pour l'évaluation de

fonctions basée sur TFHE avec plusieurs espaces de clair, nous proposons le premier cadre de synthèse permettant des transitions stratégiques entre l'espace binaire et les espaces de clair de grande dimension, en utilisant les XOR pour la logique linéaire et en concentrant les opérations non linéaires dans des LUT compactes ; (iv) pour les GC, nous introduisons le *graphe XOR-OneHot-inverter* (X1G), une nouvelle représentation intermédiaire améliorant l'efficacité des chiffrés et permettant des passes d'optimisation dédiées ; (v) enfin, nous introduisons la notion de *complexité multiplicative conjointe* (JMC), un modèle de coût conscient du coût de brouillage tenant compte du partage de propriété dans le calcul multipartite basé sur les GC, et proposons le premier optimiseur sensible à cette notion, réduisant les portes non linéaires évaluées conjointement.
Pris collectivement, ces résultats démontrent deux enseignements généraux : (1) *la représentation importe* — le choix de la forme intermédiaire influence directement les gains réalisables et l'étendue des optimisations ; et (2) *les modèles de coûts doivent correspondre à la réalité cryptographique* — une modélisation précise des goulets d'étranglement est essentielle pour obtenir des gains d'efficacité significatifs. Au-delà de ces conclusions techniques, la thèse plaide pour une standardisation renforcée et pour des interfaces inter-couches plus robustes entre bibliothèques cryptographiques, compilateurs et matériel, afin de garantir que les optimisations à un niveau restent alignées avec les avancées aux autres niveaux. En reliant la synthèse logique aux techniques modernes de calcul sécurisé, ce travail contribue à rendre les protocoles cryptographiques non seulement puissants sur le plan théorique, mais également véritablement utilisables à grande échelle.

Mots clefs : synthèse logique, conception de compilateurs pour la cryptographie, calcul sécurisé, chiffrement entièrement homomorphe, circuits brouillés.

Contents

List of Figures

List of Tables

List of Acronyms

2PC	Two-Party Computation
ACD	Ashenhurst–Curtis Decomposition
AES	Advanced Encryption Standard
AIG	And-Inverter Graph
ASIC	Application-Specific Integrated Circuit
BDD	Binary Decision Diagram
BS	Bound Set
CDC	Controllability Don't Care
CMOS	Complementary Metal-Oxide-Semiconductor
CNN	Convolutional Neural Network
CRT	Chinese Remainder Theorem
DAG	Directed Acyclic Graph
DC	Don't Care
DNN	Deep Neural Network
EDA	Electronic Design Automation
ESOP	Exclusive-or Sum of Product
FFT	Fast Fourier Transform
FHE	Fully Homomorphic Encryption
FL	Federated Learning
FMC	Functional Multiplicative Complexity
FPGA	Field Programmable Gate Array
FS	Free Set
GC	Garbled Circuit
GCD	Greatest Common Divisor
GLIFT	Gate-Level Information Flow Tracking
HDL	Hardware Description Language
ILP	Integer Linear Programming
IR	Intermediate Representation
ISA	Instruction Set Architecture
JMC	Joint Multiplicative Complexity
K-NN	K-Nearest-Neighbor

LSB	Least Significant Bit
LUT	Look-Up Table
LWE	Learning with Errors
MC	Multiplicative Complexity
MD	Multiplicative Depth
MFFC	Maximum Fanout-Free Cone
MIG	Majority-Inverter Graph
ML	Machine Learning
MLaaS	Machine-Learning-as-a-Service
MSB	Most Significant Bit
MV-PBS	Multi-Value Programmable Bootstrapping
NoC	Network on Chip
NTT	Number Theoretic Transform
ODC	Observability Don't Care
OT	Oblivious Transfer
PBS	Programmable Bootstrapping
PI	Primary Input
PO	Primary Output
PPA	Performance, Power, and Area
RAM	Random-Access Memory
RLWE	Ring Learning with Errors
RNN	Recurrent Neural Network
RNS	Residue Number System
SAT	Boolean Satisfiability
SIMD	Single Instruction Multiple Data
SMC	Structural Multiplicative Complexity
SMPC	Secure Multi-Party Computation
SMT	Satisfiability Modulo Theories
SOC	Secure Outsourced Computation
SS	Shared Set
SSV	Single–Selection Variable
TLWE	Torus Learning with Errors
TFHE	Torus Fully Homomorphic Encryption
X1G	Xor-OneHot-inverter Graph
XAG	Xor-And-inverter Graph

1 Introduction

Securing data privacy is a foundational requirement for contemporary computing and communication systems. Recent high-profile incidents illustrate the costs of inadequate protection, including the *2017 Equifax breach* that exposed the personal information of approximately 143 million individuals [96], and the *2023 MOVEit breach*, where an exploitation of the vulnerability within the *MOVEit Transfer* software resulted in hundreds of organizational breaches affecting tens of millions of people [161]. The genetic data exposure at *23andMe data leak* [97] further underscored risks unique to highly sensitive domains. These events collectively demonstrate both the scale and diversity of privacy harms that arise from compromised data.

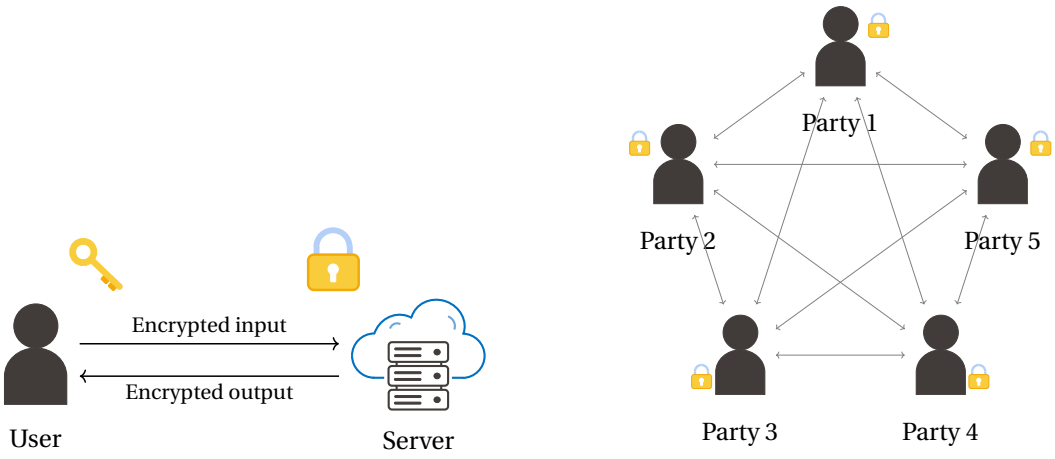
Traditional encryption (e.g., the *advanced encryption standard* (AES) [141]) is highly effective for protecting *data at rest* and *data in transit*. However, during active processing, plaintext typically must be exposed to compute on it, creating a "data in use" gap that conventional mechanisms do not directly address. As a result, while data continue to drive innovation across domains, the inherently private nature of much of this data often prevents collaborative computation even when mutual benefit is clear.

The gap created a pressing need for *secure computation*, also known as *privacy-preserving computation*.

1.1 Secure Computation Techniques

Secure computation refers to a class of techniques that enable the execution of computations on sensitive data while keeping the data itself confidential. Broadly, two paradigms organize the field:

- ***Secure outsourced computation* (SOC).** A data owner delegates computation to an untrusted evaluator (e.g., a cloud service [74]) without revealing plaintext data. Figure 1.1 a depicts a typical SOC scenario.
- ***Secure multi-party computation* (SMPC).** Multiple parties, each holding private inputs,



(a) Secure outsourced computation.

(b) Secure multi-party computation.

Figure 1.1: Secure computation paradigms. (a) *secure outsourced computation* (SOC): a single client encrypts its data locally and outsources computation to an untrusted server that evaluates directly on protected data (e.g., via fully homomorphic encryption) without access to decryption keys; only encrypted results are returned. (b) *secure multi-party computation* (SMPC): multiple parties jointly compute a function over their private inputs using garbled circuits or related primitives; messages exchanged between parties remain cryptographically protected, and only the prescribed outputs are revealed. The lock icon indicates computation over protected data. The key icon denotes possession of the decryption capability. Arrows depict data flow.

jointly evaluate a function so that nothing is learned beyond the prescribed outputs. Figure 1.1 b illustrates a collaborative setting, where five participants exchange encrypted messages to compute a shared result without revealing their private inputs.

1.1.1 Secure Outsourced Computation vs. Secure Multi-Party Computation

To illustrate the distinction between SOC and SMPC, we highlight two representative *machine learning* (ML) scenarios:

SOC for *ML-as-a-Service* (MLaaS): When a single data owner (namely, a client) wishes to offload intensive ML inference or training to an untrusted compute provider, SOC protects confidentiality by executing directly on encrypted inputs. Encryption schemes that support computation over ciphertexts allow the provider to evaluate the model without access to plaintext; the server returns only encrypted outputs, which only the client can decrypt. Early systems established feasibility for private inference [85]; subsequent work reduced latency and improved accuracy, including *hybrid* protocols that assign layers or operators to the most efficient secure primitive — mixing ciphertext-based computation with garbled circuits or secret-sharing, which are SMPC techniques, where advantageous — to lower end-to-end

cost [103, 132, 101].

SMPC for *federated learning* (FL): In cross-organizational or cross-device training, many parties collaboratively train a model without sharing raw data. A widely deployed primitive is *secure aggregation* [29], which lets a server learn only the sum of clients' updates while hiding each individual contribution. This protects local data while enabling scalable training across large client populations. Production systems demonstrate practicality at scale and motivate protocol/implementation research on communication efficiency, robustness to client dropouts, and adversarial robustness [30].

Although we contrasted two categories (i.e., SOC vs. SMPC), techniques from one can be adapted to the other. For instance, *multi-party homomorphic encryption* realizes SMPC-style tasks using SOC-native tools by enabling joint evaluation over ciphertexts with threshold decryption, thus reducing online interactivity [136, 135].

1.1.2 Performance Limitations of Modern Secure Computation Techniques

The two broad styles of modern secure computation tend to hit very different bottlenecks in practice.

SOC techniques: computation and memory are the limiters. In SOC tasks, a single untrusted server (or a small set of servers) evaluates the function on encrypted inputs returned to the client at the end. With *fully homomorphic encryption* (FHE) [155], this evaluation is performed directly on ciphertexts. The required arithmetic (e.g., large-degree polynomial operations and periodic "refresh" steps to control encryption noise) remains orders of magnitude slower than ordinary arithmetic on plaintexts. Ciphertexts and keys are also much larger than their plaintext counterparts, so moving them through the memory hierarchy can become the dominant cost. Overall, FHE workloads tend to be *compute- and memory-bound*, not network-bound.

SMPC techniques: communication is the limiter. Protocols in which several parties jointly compute on their private inputs typically require the parties to exchange information repeatedly as the computation proceeds. Even when the number of exchanges ("rounds") is kept small, the *volume* of data that must be transmitted can be large. A prominent example is the *garbled circuit* protocol (GC) [190]: it compresses interaction into a few rounds, but each non-linear gate of the circuit is represented by encrypted table entries that must be sent over the network. In both cases — many small exchanges (which is the case of the *secret sharing* protocol [159]) or a few very large transfers — end-to-end performance is dominated by network latency and bandwidth rather than local computation.

The boundary is blurry, but the bottlenecks persist. As briefly mentioned before, techniques are often adapted across settings. For example, FHE can be used in multi-party protocols (e.g., threshold [9] and multi-party variants [136]), and, conversely, the GC protocol can be tuned to facilitate outsourcing computation privately [45, 84]. However, the dominant cost usually follows the primitive being used: FHE-based SMPC protocols remain limited by heavy encrypted computation and memory traffic, whereas SOC protocols built from GC or other SMPC-native techniques remain limited by data movement over the network.

Scope of this thesis. Given these performance bottlenecks, improving secure computation is both urgent and consequential. The community is attacking the problem across the stack. Before surveying this landscape, we briefly introduce the two workhorses we focus on in this thesis: *fully homomorphic encryption* (FHE), which enables computation over ciphertexts but is dominated by heavy encrypted arithmetic and memory traffic, and the *garbled circuits* protocol (GC), which evaluate Boolean circuits via encrypted lookup tables and are chiefly limited by data transfer. From this point forward, we reason directly in terms of FHE and GC (rather than the broader SOC/SMPC labels) and tailor our contributions to their characteristic performance profiles. Formal definitions and technical details for both will be introduced in Chapter 2.

1.1.3 Fully Homomorphic Encryption

Origins. The idea of computing on encrypted data traces back to *privacy homomorphisms* [155] and early *partially* homomorphic schemes (e.g., systems that support either only additions or only multiplications on ciphertexts).

Early general constructions. A full solution arrived with Craig Gentry [82]: a construction over *ideal lattices*[1] that supports both operations and thus arbitrary circuits. Hiding a message by adding a small random *error* to a linear relation with a secret key, the security rests on the presumed hardness of certain lattice problems. Homomorphic operations transform ciphertexts and their error (a.k.a. noise):

- additions add the noises (and are therefore gentle),
- multiplications grow noise much faster (due to cross terms and rescaling).

Thus, without intervention, the noise eventually exceeds the decryption radius, breaking correctness. Correspondingly, Gentry also introduced *bootstrapping*: homomorphically evaluating (a low-depth version of) the decryption circuit to *refresh* a ciphertext's noise so further computation remains possible.

[1] Very informally, a lattice is a discrete additive subgroup of $\mathbb{R}^n$; an ideal lattice is a lattice whose basis vectors arise from a ring ideal.

Shortly after, integer-based designs by Marten van Dijk et al. [73] showed FHE can be built without polynomial rings — the plaintext is a bit, ciphertexts are noisy integers, and homomorphic operations are just addition and multiplication of integers. Security relies on the *approximate greatest common divisor* (approximate-GCD) problem: given many integers of the form $x_i = p \cdot q_i + r_i$, where the secret p is fixed and the "errors" r_i are small, it should be hard to recover p. Despite being conceptually simple, it is far too slow for practical workloads — to achieve even modest security, p must be enormous, resulting in prohibitively large ciphertexts and, therefore, correspondingly slow homomorphic operations.

- **Leveled schemes.** These schemes avoid bootstrapping during evaluation by budgeting a maximum *multiplicative depth* (MD) in advance. They are typically built from the *learning with errors* (LWE) assumption: given linear equations modulo q with small random noise added, recovering the hidden vector is presumed hard. The *ring-LWE* (RLWE) variant moves from vectors to polynomials, which accelerates arithmetic via the *number theoretic transform* (NTT) and yields compact keys and ciphertexts. Representative schemes include *BGV* [35] and *BFV* [81] for exact modular arithmetic. They control noise using tools such as *modulus switching* / *rescaling* (lowering the modulus to shrink noise relative to ciphertext scale) and *relinearization* / *key switching* (keeping ciphertext size small after multiplications). *CKKS* [52] adds *approximate* real/complex arithmetic: ciphertexts encode fixed-point vectors and operations track small rounding errors, which is ideal for ML inference and analytics that tolerate approximation.

- **Fast-bootstrapping schemes.** *FHEW* [75] showed sub-second bootstrapping for binary gates, and *TFHE* [57] pushed this further with highly optimized *programmable bootstrapping* (PBS): a refresh that simultaneously evaluates a small function, enabling efficient Boolean logic and small-integer operations.

Features and practice today. Across these families, several patterns recur:

- *Packing* RLWE-based schemes can pack many plaintext values into one ciphertext (via the Chinese Remainder Theorem on cyclotomic rings), enabling *single-instruction–multiple-data* (SIMD) parallelism.

- *Noise growth and refresh.* Every homomorphic operation increases noise; leveled schemes manage growth until the pre-set depth is exhausted, while fast-bootstrapping schemes refresh frequently to reset noise.

- *Workload fit.* BFV/BGV are preferred for exact modular arithmetic; CKKS dominates approximate, real-valued workloads (e.g., private inference); FHEW/TFHE excels on Boolean circuits and small-integer computations.

1.1.4 Garbled Circuits

Originally proposed by Andrew Yao for *secure two-party computation* (2PC) [190], the GC protocol turns a Boolean circuit for a function into an encrypted artifact that one party ("garbler") prepares and the other ("evaluator") executes. At a high level, the garbler assigns two random labels to every wire (one per logical value), encrypts each gate's truth table using the input labels as keys, and sends the resulting "garbled tables" plus the appropriate input labels to the evaluator. The evaluator obtains its own input labels through a standard private retrieval subroutine (*oblivious transfer* (OT)), then evaluates the garbled circuit gate-by-gate without learning intermediate plaintexts. GC's round complexity is constant — communication happens in a small number of bursts independent of circuit depth — which has made it especially attractive over wide-area networks.

Milestone extensions. Over nearly four decades, several key ideas made GC practical at scale:

- **From 2PC to MPC.** The *Beaver–Micali–Rogaway* (BMR) [18] paradigm generalized Yao's two-party design to multi-party settings by letting several parties jointly "share" the garbler role while still producing a single garbled circuit that any designated evaluator can run.

- **Lower communication per gate.** Techniques such as *garbled row reduction* (eliminating one ciphertext per garbled table) [139], *free-XOR* (making XOR gates cost-free by a global label offset) [108], and *half-gates* (garbling an AND gate with just two ciphertexts, which is proved to be the optimum) [197] dramatically cut table sizes.

- **Cheating resistance.** *Cut-and-choose* style compilers and related checks [117, 116] lift GC from the *semi-honest model* (parties follow the protocol) to *malicious security* (parties may deviate), trading extra work for stronger guarantees.

- **Systems advances.** OT extension (amortizing multiple oblivious transfers) [107], vectorized, AES-based implementations [20], and optimized circuit libraries (e.g., for comparisons, arithmetic, and cryptographic primitives) [72, 143] pushed GC into the million-gate regime.

What GC is good at today. GC is a leading general-purpose tool when the target computation is naturally Boolean or when one wants constant-round protocols. It underpins private set operations, privacy-preserving analytics, auctions, and components of secure machine-learning inference and training [143]. Mature software stacks and hybrid frameworks combine GC with other techniques (e.g., arithmetic secret sharing [150]) to match workload structure.

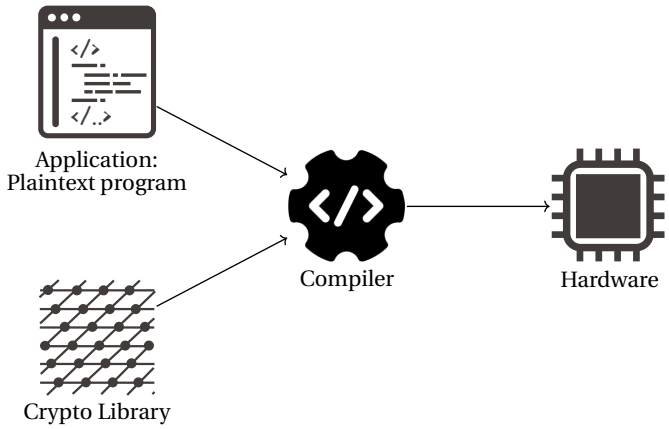


Figure 1.2: Ongoing full-stack efforts to make FHE-based secure computation practical.

Where costs come from. GC's performance is dominated by communication: every non-linear gate contributes encrypted table entries that must be transmitted, while XOR is essentially free under modern optimizations. As a result, improving GC typically means either

(i) synthesizing circuits that use fewer costly gates, or

(ii) adopting garbling schemes that reduce bytes per gate.

Both themes we will return to in later chapters (Chapters 6–7).

1.2 Research Efforts for Accelerating Secure Computation: The Case of Fully Homomorphic Encryption

Despite the theoretical appeal and growing maturity, secure computation techniques still suffer from substantial practical limitations — most notably, the *computational* and *communicational* overhead compared to plaintext computation. These overheads remain a central barrier to adoption. They span orders of magnitude and manifest across various system levels. Consequently, a wide range of research efforts has emerged to mitigate this overhead, spanning the full system stack — from *algorithm design* to *cryptographic primitives*, *compiler design* and *hardware acceleration*.

To illustrate the diversity and structure of these efforts, this section presents a layered view of optimization strategies, using FHE as a representative example, as Figure 1.2 depicts. FHE is selected here due to its broad applicability, strong theoretical foundations, and significant progress in both academic and industrial contexts. Nevertheless, the layered framework discussed below applies, with minor adaptation, to the GC protocol as well.

1.2.1 Application-Level Optimization

At the application layer, the strongest gains come from co-designing the task specifically for FHE. A common pattern is to start with a plaintext algorithm or model and address components that are poorly matched to homomorphic arithmetic (e.g., comparisons, divisions, softmax) by:

- *refactoring* them into low-degree polynomial approximations or algebraic surrogates that preserve accuracy while keeping multiplicative depth low; or
- *hybridizing* with a complementary secure-computation primitive so that each subroutine runs under the most cost-effective protocol.

Classical Statistical Learning with FHE Early work mapped classical statistical learners (e.g., logistic regression, linear models, naïve Bayes) to FHE using *fixed-point encodings* and *polynomial surrogates* for divisions and comparisons. These early techniques laid the foundation for more advanced applications, particularly in tabular and medical analytics [11].

Extending to Neural Networks Subsequent efforts adapted the same principles to modern ML. `CryptoNets` demonstrated end-to-end private *convolutional neural network* (CNN) inference by retraining networks with *low-degree* activation polynomials and exploiting aggressive *SIMD batching* under leveled FHE [85]. `CryptoDL` further refined this approach by systematizing activation approximation and *data-layout* refactoring to balance accuracy, depth, and rotation counts [79]. To extend to *recurrent neural network* (RNN) models, researchers have proposed leveled FHE implementations of gated RNNs with *manually optimized MD* per recurrent step and restrained sequence length through *architectural rearrangements* [182]. Such designs avoid the costly use of bootstrapping and maintain model accuracy across common benchmarks, facilitating practical applications such as *private text classification* [183]. More recently, FHE-compatible Transformer inference has been enabled by simultaneously reducing attention dimensionality and replacing softmax/normalization with *low-degree* polynomial surrogates [49].

Notably, across these neural architectures, *approximated* leveled schemes, such as CKKS, have become the method of choice due to their SIMD-friendly computation model and tolerance to numerical noise — traits that align well with the data-massive and fault-tolerant natures of NN models. For the same reason, combining precise leveled schemes (e.g., BGV or BFV) with neural network approximation techniques, such as *model quantization* [113], or adopting approximate representations on both the cryptographic and model sides [47], has also proven to be an effective strategy in practice. In contrast, FHE solutions based on fast-bootstrapping schemes remain less common, though promising advances continue to emerge [54, 61].

Protocol Co-Design for Efficiency A complementary line of work improves performance by co-designing the computation protocol. The shared design principle is to retain high-throughput linear algebra under FHE and implement branching and comparisons under protocols better suited for discrete operations.

`Gazelle` combines FHE with GC for *deep neural network* (DNN) inference, keeping *linear* layers in FHE while offloading *discrete* nonlinearities to GC [103]. `Delphi` optimizes this partitioning to further reduce latency [132], and `CrypTFlow2` generalizes this hybrid composition by integrating arithmetic secret sharing, FHE, and GC to support diverse ML pipelines [147].

Recent work shows that *cross-protocol partitioning* of Transformer layers — not simply replacing each plaintext layer with its FHE equivalent — can reduce ciphertext rotations and expensive inter-protocol conversions, leading to substantial performance improvements [187].

1.2.2 Cryptographic Construction-Level Optimization

At the cryptographic-construction layer, performance is dictated by how a scheme realizes arithmetic on ciphertexts and by the efficiency of its supporting subroutines. Below, we summarize the main optimization families and how modern libraries expose them.

Parameterization and security. Schemes surface tunables such as the polynomial degree n, ciphertext modulus q, and plaintext modulus p. Choosing *just-large-enough* parameters is critical: increasing n or q typically raises cost super-linearly (NTT sizes, key-switch dimensions, memory footprint) and can weaken security if not jointly sized. In practice, libraries couple standardized security guidance for (R)LWE (e.g., the *HomomorphicEncryption.org standard* [1] and the *LWE estimator* [2]) with pragmatic noise-growth models to right-size depth and precision. Mis-sizing remains a recurrent source of overhead: overly conservative chains waste time and memory; overly aggressive ones force re-designs or bootstraps.

Leveled schemes vs. fast-bootstrapping schemes. Leveled schemes (BGV/BFV/CKKS) select parameters so an entire circuit runs without bootstrapping; maintenance consists of relinearization and modulus/scale management. By contrast, fast-bootstrapping schemes (FHEW/TFHE) embrace frequent *programmable bootstrapping* (PBS) to refresh noise and optionally evaluate a small function during the refresh. Some leveled stacks now include improving bootstraps, but TFHE-style families are architected around sub-100ms PBS in optimized builds.

Leveled schemes: encoding, automorphism, and ciphertext maintenance. For BFV/BGV, the *Chinese Remainder Theorem* (CRT) packing turns the plaintext ring into many independent "slots" [92]; similarly, CKKS encodes complex vectors at a given scale. While packed ciphertexts enable slot-wise SIMD, aligning data across slots requires *automorphisms* (rotations), each

realized via key switching. Consequently, data layout (how tensors are packed) is often the largest single source of speedup [26].

After multiplications, ciphertext "degree" and noise grow. Three maintenance primitives control this growth: *relinearization* (degree reduction via key switching), *modulus switching* (BGV/BFV), and *rescaling* (CKKS) to keep scale stable. When to apply them is a major scheduling choice: eager maintenance simplifies correctness but wastes work; delayed or fused maintenance (e.g., defer last relinearization, align rescale boundaries with algorithmic layers) routinely yields double-digit speedups [93, 51].

Fast-bootstrapping schemes: table-lookup style computation. In FHEW/TFHE, PBS not only refreshes but also evaluates a small function — classically a Boolean gate or a *lookup table* (LUT). Performance rests on highly tuned PBS kernels (*fast Fourier transform* (FFT)/NTT backends, cache-aware gadget decompositions [57, 140]) and on reducing the *count* of PBS calls with higher-arity operations [123] and LUT fusion [43]. These features contribute to end-to-end speedups for Boolean and small-integer workloads.

Library ecosystems and exposed controls. Mature stacks package these techniques behind ergonomic APIs while exposing expert knobs for compilers: `HElib` (BGV/BFV; rotation hoisting, relinearization controls) [93], `SEAL` (BFV/CKKS; *residue number system* (RNS)/NTT kernels, scale/modulus management) [51], `PALISADE/OpenFHE` [14] and `Lattigo` [111] (BGV/BFV/CKKS; auto parameter selection and key management), and `TFHE-rs/Concrete` (PBS; FFT/NTT backends) [199, 198].

Meanwhile, thanks to sustained cryptographic research, these libraries continue to evolve with increasingly performant kernels. For instance, notable progress has been made in accelerating bootstrapping operations across both leveled and fast-bootstrapping FHE schemes [94, 32, 55].

Typical controls exposed by the libraries include modulus-chain builders, encoder/packing toolkits, rotation-key planners (for leveled schemes), and PBS planners (for fast-bootstrapping schemes) — precisely the hooks that a compiler exploits to translate application structure into efficient encrypted circuits.

1.2.3 Compiler-Level Optimization

Even with mature FHE libraries, writing performant applications remains a specialist task: depending on the target FHE scheme, developers must select secure yet tight parameters, place ciphertext–maintenance operations (e.g., relinearization, rescaling/modulus switching), manage data layout and rotations for SIMD batching, and control multiplicative depth while preserving correctness. FHE compilers raise the abstraction: they accept a high-level program, lower it through one or more *intermediate representations* (IRs), and automatically and op-

timally insert scheme-specific operations and data movement to produce efficient kernels targeting software or hardware backends.

Existing tools show two families:

- general-purpose compilers (e.g., `ALCHEMY` [66], `RAMPARTS` [7], `HECO` [177], `HEIR` [3]), and
- ML-oriented compilers (e.g., `nGraph-HE` [25], `SEALion` [80], `Orion` [76], `ANT-ACE` [115]).

A layered compiler view. A useful organizing principle is a multi-stage pipeline:

(i) *Program transformations* that restructure high-level code into the primitives provided by the target cryptographic library;

(ii) *Circuit optimizations* that transform a naïve realization of the program (a circuit) into a functionally equivalent, lower-cost one;

(iii) *Cryptographic optimizations* that choose parameters and place maintenance (and, where available, bootstrapping) to meet security and accuracy at minimal cost;

(iv) *Device-aware scheduling* that lowers to CPUs/GPUs or emerging accelerators.

This end-to-end view motivates distinct IRs, each exposing just enough structure to enable the right optimizations at that level. A visualization of the pipeline is offered in Figure 1.3.

Program transformations (front-end). Compilers first normalize control flow, inline or flatten loops with static bounds, and canonicalize expressions. For leveled schemes, they aggressively *batch* work to exploit SIMD slots, but — because FHE supports only restricted data movement (primarily cyclic rotations) — automatic batching must reason *globally* about index patterns rather than perform purely local vectorization. `HECO` [177], for example, unrolls statically sized loops, merges associative chains into n-ary operations, and introduces a *BatchedSecret* abstraction; it replaces elementwise arithmetic with whole-vector operations and aligns operands via rotations chosen by a *target-slot* heuristic, enabling a clean-up pass to eliminate redundant rotations/extracts. A dedicated *rotate-and-sum* pass captures common ML and stencil idioms, reducing multiplicative depth. The prevalence of domain patterns at this stage also motivates application-specific compilers for ML, which can surface additional structure for later passes.

Circuit optimizations (middle-end). Once mapped to homomorphic operations, the program is a circuit. For leveled schemes (additions and multiplications as the basic elements), algebraic rewrites — re-association into balanced trees, common-subexpression elimination, and sharing exposure — reduce the number and span of multiplications (and thus

multiplicative depth). For fast-bootstrapping schemes (e.g., TFHE), where programmable bootstrapping (PBS) evaluates table lookups, the compiler synthesizes a compact LUT network that realizes the target function with as *few* LUTs (PBS calls) as possible. These passes, while individually modest, compound with parameter planning. They are standard across tools such as `Cingulata` (which also leverages Boolean-level EDA optimizers) [42] and `CHET/EVA` (graph rewrites and maintenance placement for CKKS) [69, 70].

Cryptographic optimizations (parameter and maintenance planning). Choosing parameters (polynomial modulus degree, coefficient moduli, scaling strategy) is both correctness- and performance-critical. For leveled schemes, compilers estimate noise growth to select the smallest sound parameters and to *schedule maintenance* (e.g., relinearize after certain multiplies, rescale/mod-switch at suitable cut points). When bootstrapping is available, recent planners treat it as a costed operation and insert it only when it reduces overall latency or memory [185, 53, 119]. For fast-bootstrapping schemes, PBS plays both roles — function evaluation and noise refresh — so explicit maintenance placement largely disappears; the focus shifts to LUT synthesis and PBS key management.

Device-aware scheduling (backend). Finally, compilers lower to CPU libraries (e.g., `SEAL` [51]) or to low-level polynomial/RNS dialects suitable for vector *instruction set architectures* (ISAs) (e.g., `AVX2/AVX-512`) and accelerators. Register/memory scheduling, kernel fusion, and ISA-aware loop transformations matter because ring-NTT pipelines and RNS decompositions dominate runtime. Recent modular IRs enable specialization per backend and pave a path to heterogeneous targets [110].

In summary, the compiler's role is to harmonize application intent with the capabilities and constraints of a chosen FHE library, automatically exploiting batching and data layout, reducing multiplicative depth and expensive operations, and selecting parameters and schedules that satisfy security/accuracy while minimizing end-to-end cost, while the specific tasks are scheme-dependent.

1.2.4 Hardware-Level Optimization

At the hardware layer, accelerating FHE hinges on two broad levers:

(i) exploiting *data parallelism* in the heavy primitives (NTT/FFT, modular arithmetic, key switching, and bootstrapping), and

(ii) *co-designing* microarchitectures around scheme-specific kernels (e.g., PBS for fast-bootstrapping schemes, rescale/relinearize for leveled schemes).

We review three complementary directions — CPU vectorization, GPU offloading, and dedicated FHE accelerators — emphasizing their roles, trade-offs, and representative systems.

CPU vectorization (SIMD on general-purpose cores). Modern CPUs support wide SIMD extensions such as Intel `AVX2` and `AVX-512` (advanced vector extensions), which allow the same arithmetic/logical operation to be applied across many integers in parallel. FHE libraries increasingly exploit these lanes in their low-level kernels (NTT butterflies, modular multiplication, etc.), often via intrinsics. A canonical example is Intel's open-source `HEXL` library, which provides `AVX2/AVX-512` vectorized primitives that several higher-level libraries (e.g., `SEAL` and `PALISADE/OpenFHE`) use to speed up ring-NTTs and modular reductions [24]. Vectorization does not require specialized device, portable across servers, and benefits from mature compiler backends; however, memory bandwidth and the width of the SIMD unit cap throughput, and CPUs cannot match the massive thread-level parallelism of GPUs.

GPU acceleration (massively parallel kernels). GPUs offer thousands of lightweight cores and high memory bandwidth, a natural fit for the batched, regular dataflow in NTT/FFT and key-switching. Using `CUDA` or `HIP` (programming models for GPUs, developed by NVIDIA and AMD, respectively), several efforts implement scheme backends directly on GPUs. For TFHE-style PBS, the `cuHE` line of work demonstrated order-of-magnitude speedups by mapping blind-rotation and external product to FFT/NTT kernels on GPUs [68]. For approximate CKKS, recent systems such as `Cheddar` target end-to-end GPU execution and report large gains over CPU baselines by careful kernel fusion and memory hierarchy tuning [58]. GPUs provide excellent *throughput* and decent *latency* for large batches, with strong software ecosystems. Their limitations include host–device movement, contention in multi-tenant environments, and the need to re-tune kernels across device generations for peak performance.

Dedicated accelerators. When latency, energy, or scale dominate, dedicated accelerators let designers co-optimize datapaths, memory tiling, and arithmetic widths around the FHE workload. These designs can be categorized along two axes: whether they target leveled or fast-bootstrapping schemes, and whether they specialize in optimizing individual kernels or accelerate end-to-end computation.

Kernel-level accelerators for leveled schemes. When full programmability is unnecessary, specialized designs offer tighter optimization. Wang et al. focus on RNS-CKKS key switching, implementing a pipelined *application-specific integrated circuit* (ASIC) design that avoids frequent *random-access memory* (RAM) access and allows tunable parallelism for higher throughput [181]. Pont et al. target non-power-of-two NTTs in BGV by combining *prime-factor* and *rader* algorithms, achieving up to 5× area efficiency over prior circuit designs [146].

End-to-end accelerators for leveled schemes. Roy et al. propose an Arm+*field programmable gate array* (FPGA) co-processor for BFV-based FHE, using pipelining and parallelism to achieve

a 13× speedup over optimized CPU software [162]. Microsoft's HEAX architecture exemplifies a domain-specific design targeting leveled schemes, featuring replicated NTT/multiplication tiles and on-chip buffers for ciphertext residues to reduce latency and improve key-switching throughput [151]. F1 extends this idea with a programmable wide-vector architecture and compiler-driven data movement scheduling to support general FHE programs with high throughput [157]. BTS introduces bootstrapping as a first-class citizen, proposing a tiled grid of processing elements and a communication-aware *network on chip* (NoC) to deliver high performance on workloads such as ResNet inference and logistic regression under leveled FHE schemes [104]. CraterLake demonstrated an ASIC for *unbounded* FHE computation by accelerating bootstrapping itself, showing large speedups and energy savings over prior software and reconfigurable baselines [156].

Kernel-level accelerators for fast-bootstrapping schemes. Bertels et al. accelerate the FHEW bootstrapping kernel via an optimized NTT module for negatively wrapped convolutions, achieving 7.5× speedup with manageable LUT and DSP footprints on FPGA [23].

End-to-end accelerators for fast-bootstrapping schemes. MATCHA targets the same setting with a pipelined architecture based on multiplication-free FFT approximations to minimize latency and energy [102]. FPT reinterprets torus arithmetic in fixed-point and implements a datapath on FPGAs that enables 35μs bootstrapping, further enhancing throughput without compromising accuracy [175].

This growing body of work reflects an ongoing architectural evolution tailored to the distinct performance bottlenecks of each FHE family — whether by building deeply pipelined datapaths or tightly tuned engines for kernels.

While the above discussion focuses on FHE as a representative case study, similar full-stack optimization principles apply to the GC protocol. Though the concrete challenges and cryptographic primitives differ between FHE and GC, the overarching architectural view and the need for end-to-end co-design remain consistent.

1.3 Research Scope

1.3.1 Role of Logic Circuit Optimization

Where this thesis sits in the stack. Figure 1.3 positions this thesis squarely inside the *circuit optimization* stage (middle-end) of a full FHE compilation flow. While the figure illustrates the specific case of an FHE compiler, the process for GC follows a similar structure, and our GC-related contributions also reside in this middle layer. Concretely, despite the secure computation scheme, we assume the target task is a generic Boolean transformation $f : \mathbb{B}^n \mapsto \mathbb{B}^m$ that is not subject to certain applications, and we operate on its *Boolean circuit* implementation. Then, based on the chosen secure computation scheme and its software library (the "crypto backend"), our goal is to build a *multi-level logic network* whose primitive

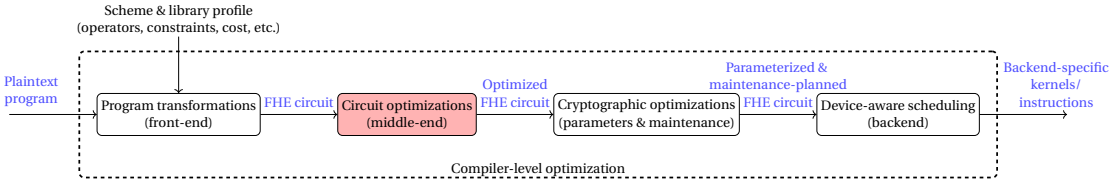


Figure 1.3: An end-to-end FHE compilation pipeline. Each box denotes a compiler stage. Arrows labeled in blue indicate the evolving artifact: from a high-level plaintext program, to an FHE circuit (homomorphic operator graph), to an optimized FHE circuit, then a parameterized and maintenance-planned FHE circuit, and finally backend-specific kernels/instructions (CPU/GPU/accelerator). The scheme/library profile (available operators, constraints, etc.) feeds into the front-end. The contribution of this thesis focuses on the circuit-optimization stage (highlighted in red).

nodes are the library's supported operations (e.g., homomorphic addition and multiplication in the case of leveled FHE schemes), such that the network realizes f while minimizing a scheme-specific cost metric. In this sense, *each protocol induces its own logic representation and cost model*, and therefore its own optimization problem: what the "gates" are and what they "cost" are determined by the cryptographic construction.

A brief look back: logic synthesis in its native CMOS setting. To appreciate the novelty and boundaries of our contributions, it is helpful to recall the classical role of *logic synthesis* in *complementary metal-oxide-semiconductor* (CMOS)-based digital design. Within *electronic design automation* (EDA), logic synthesis maps high-level specifications to optimized gate-level netlists, with quality measured by *performance*, *power*, and *area* (PPA). Specifically, traditional flows emphasize minimizing gate count, depth (critical-path delay), and switching activity [71]. Two trends have recently renewed attention on logic-level optimization:

(i) As transistor scaling slows, fewer PPA gains come "for free" from process technology, so synthesis must deliver more; and

(ii) *Emerging devices* bring non-standard constraints that must be enforced at the logic level (e.g., path balancing in superconducting electronics to ensure correct timing of *single-flux-quantum pulses* [15]).

In short, logic synthesis evolves whenever the *target substrate* (CMOS or otherwise) changes the rules of what costs matter and what structures are permissible.

Why secure computation needs its own logic optimization. This thesis carries the logic-synthesis mindset into the cryptographic substrate. In our setting, the "technology library" is a secure computation scheme; its "cells" are cryptographic primitives; its "timing" and "area" translate into protocol-level costs such as ciphertext generation, transmission, and homomorphic evaluation latency.

1.3.2 Contributions of This Thesis

Prior efforts have already leveraged this perspective to optimize circuits for secure compution:

- **Fully homomorphic encryption (FHE).** In the circuit optimization problem for leveled schemes (e.g., BFV/BGV), cost is dominated by multiplicative depth [42, 41, 12, 112]. In fast-bootstrapping schemes (e.g., TFHE), computation proceeds via programmable bootstraps (LUTs), making the *number of PBS calls* the primary driver, whose circuit optimization requirement can be interpreted as an *area-oriented LUT mapping problem* [89, 86].
- **Garbled circuits (GC).** Linear gates (XORs) are essentially free when being garbled under *free-XOR* [108], whereas non-linear gates incur *garbling cost* (ciphertexts to be generated, transmitted, and evaluated), which motivates formulating the corresponding circuit optimization problem as *reducing the multiplicative complexity of XOR-AND-inverter graphs* [166, 152, 172, 171].

These cost models underpin much of the literature on circuit optimization for secure computation. However, as the broader stack evolves (see Figure 1.2) — with advances in cryptographic constructions, compiler analyses, and hardware support — the "right" objective at the circuit level may shift. This thesis revisits those assumptions: we

(i) re-examine scheme-specific logic *representations* (intermediate forms),

(ii) refine the *cost models* that truly capture today's protocol bottlenecks, and

(iii) design *scalable algorithms* that optimize against these updated objectives.

In short, we close a gap between traditional circuit proxies and the realities of modern secure computation.

A detailed, chapter-by-chapter summary follows in the next section on thesis organization.

1.4 Thesis Organization

The remainder of this thesis is organized into a foundations part (Chapter 2) and two methodologically focused parts: Part I develops logic-level techniques for accelerating FHE, and Part II advances GC generation through new representations and cost models. Each technical chapter is self-contained, but dependencies are indicated where results build on earlier material.

Chapter 2 (Preliminaries). We introduce the mathematical and systems background used throughout the thesis. The chapter covers Boolean logic and logic-network formalisms, secure computation techniques at a high level (with brief, self-contained overviews of FHE

and GC families), and the specific optimization lenses we adopt. It also reviews the *logic synthesis preliminaries* relevant for later chapters (cuts, Boolean classification, and algebraic vs. Boolean-aware optimizations, etc.), establishing notation and cost models used thereafter.

1.4.1 Part I: Efficient Fully Homomorphic Encryption from the Ground Up

Chapter 3 (Beyond Depth: Multiplicative Depth vs. Complexity in Leveled FHE). In leveled FHE (e.g., BFV/BGV), the *multiplicative depth* (MD) — the length of the longest chain of ciphertext multiplications — largely dictates parameter selection. A higher MD forces larger rings and modulus chains to keep noise below the decryption threshold, which increases runtime and memory [42]. For this reason, most prior circuit-level optimizations target *only* MD minimization. However, *multiplicative complexity* (MC) — the total number of ciphertext multiplications — also strongly impacts performance, since a ciphertext multiplication is orders of magnitude more expensive than addition. As in classical logic synthesis (area–delay tension), aggressively lowering MD often inflates MC; in practice, MD-only flows commonly enforce a crude cap on MC growth and halt once the cap is hit, yielding suboptimal designs [41, 12, 112].

This chapter addresses the joint optimization problem head-on. We

(i) formalize the MD–MC trade-off for leveled schemes and adopt a composite objective guided by an empirically derived Pareto frontier;

(ii) develop an exact synthesis engine for small/medium cuts that efficiently returns *provably optimal* implementations under this objective; and

(iii) design a scalable, Boolean-correct rewriting framework that is *simultaneously* MD- and MC-aware, reducing peak multiplication depth without gratuitously increasing the multiplication count.

On standard benchmarks we report consistent MD and MC reductions, corresponding parameter savings, and tangible end-to-end improvements in both circuit synthesis time and homomorphic evaluation latency.

Chapter 4 (Technology Mapping for TFHE: Gate-Set Design and Multi-Value PBS). Chapters 4–5 both target fast-bootstrapping FHE for Boolean computation; this chapter considers the *single–plaintext-space* setting. In TFHE, a *programmable bootstrap* (PBS) can realize any two-input Boolean function in one refresh [57], which motivated prior work to cast circuit generation as an area-oriented two-input-LUT mapping problem and to reuse off-the-shelf EDA flows [89, 86]. Subsequent theoretical results showed that enlarging the plaintext space enables *higher-arity* LUTs per PBS — reducing the total number of PBS calls at the cost of a slower per-PBS kernel — often yielding net speedups for end-to-end evaluation [31, 123].

This enlarges the design space: given a single (globally chosen) plaintext space, how do we best pack logic into PBS operations so that the reduction in PBS count outweighs the per-PBS latency increase?

A further complication — and opportunity — comes from *multi-value PBS* (MV-PBS) [43], which amortizes PBS cost across multiple LUTs that *share the same inputs*. To fully capitalize on MV-PBS, the logic synthesis must not only raise LUT size where profitable, but also *shape* the network so that many LUTs reuse identical fan-ins.

This chapter develops both theory and tooling to meet that goal. On the theory side, we give a constructive strategy for stuffing larger-input functions into a single PBS by exploiting Boolean *symmetry* (output depends only on Hamming weight) and *negacyclicity* (structure induced by the polynomial ring domain), which together increase the set of functions representable within a fixed LUT size. On the engineering side, we devise an MV-PBS–aware LUT mapper that jointly

(i) selects LUT arities under a chosen ciphertext space, and

(ii) maximizes input sharing across LUTs to trigger MV-PBS amortization.

Across standard Boolean benchmarks, the resulting TFHE circuits use *fewer PBS calls* and exhibit *lower end-to-end latency* than prior mappers that exploit large LUTs and MV-PBS insufficiently [90, 40].

Chapter 5 (Encoding Strategy Management for TFHE: An ESOP-Guided Approach). This chapter also targets fast-bootstrapping FHE for Boolean computation, but from a different angle than Chapter 4. Fast-bootstrapping schemes (e.g., TFHE) can operate in a *leveled* fashion: when the plaintext space is binary, XOR can be realized by (cheap) ciphertext addition, whereas AND corresponds to ciphertext multiplication. As in leveled-FHE schemes, however, ciphertext multiplication is far more expensive (and noise-inflating) than addition; XOR alone is not functionally complete. Hence, a pure leveled-binary approach is rarely competitive [56].

A recent insight changes the picture: a PBS can also serve as a *domain switch*, converting between plaintext spaces [28]. This enables a dual-space strategy: exploit the binary space when linear operations (XOR) dominate, and jump — via PBS — into a larger plaintext space where more expressive non-linear functions are realized efficiently as LUTs, then return to binary as needed. The synthesis question becomes: how should a TFHE circuit be *structured* so that these space transitions occur at the most advantageous cut points, maximizing linear work in binary while minimizing PBS and conversion overhead?

This chapter provides the first systematic answer. We introduce a framework for *ciphertext encoding–space switching* that orchestrates computation across (at least) two encoding domains:

(i) a formal semantics with correctness conditions for inter-space conversion;

(ii) a planner that places switches and selects LUT granularities to minimize the total cost (both PBS calls for conducting computation and for switching encoding-space) under the homomorphic evaluation latency models; and

(iii) an implementation showing end-to-end latency gains on composite workloads when mixing spaces, compared to single-space baselines.

Conceptually, Chapter 4 optimizes *within* a fixed plaintext space, whereas Chapter 5 enables *cross-space* design choices, leveraging binary for cheap linear algebra and larger spaces for compact non-linear evaluation, coordinated by PBS-based switching.

1.4.2 Part II: Toward Practical Garbled Circuits

Chapter 6 (Ciphertext-Efficient Garbled Circuits via XOR-OneHot Graphs). In the *garbled circuits* (GC) protocol, the *free-XOR* technique renders linear gates (XORs) essentially free — no garbled tables, hence no communication [108]. Accordingly, most prior work models GC-oriented synthesis as an *MC reduction* problem: realize the target function as an *XOR–AND–inverter graph* (XAG) and minimize the number of AND gates [166, 152, 172, 171]. This chapter re-examines that premise: while AND is a basic carrier of nonlinearity, it is not necessarily *ciphertext-optimal* once translated to GC. We analyze the expressive power and garbling cost of small primitives under modern garbling techniques (e.g., *GRR3* [139], *garbling gadget* [16], etc.) and show that a three-input *OneHot* gate offers superior cost-per-nonlinearity. Motivated by this, we introduce the *XOR–OneHot–inverter graph* (X1G) as a ciphertext-efficient representation for GC generation. To make X1G practical, we

(i) devise an optimal, algebraic mapping from XAGs to X1Gs that preserves prior MC-reduction gains while exploiting OneHot efficiency; and

(ii) develop X1G-specific optimization algorithms that unlock additional savings unique to the representation.

Together, these techniques substantially reduce the communication cost of the resulting garbled circuits.

Chapter 7 (Redefining Cost in Garbled Circuits: Joint Multiplicative Complexity). This chapter starts from a simple but consequential observation: after a Boolean network (e.g., an XAG) is synthesized, *not every gate must be garbled.* Any gate whose value depends solely on one party's inputs can be evaluated locally in plaintext; only gates whose evaluation *jointly* depends on multiple parties need secure GC evaluation. To make this actionable, we formalize

node ownership and introduce *joint multiplicative complexity* (JMC): the number of non-linear gates (ANDs) that cannot be derived by any single party and therefore incur garbling cost.

Building on these notions, we present the first automated, general-purpose, gate-level optimization framework that *reduces JMC*. The framework labels ownership throughout the netlist, identifies jointly derived regions, and applies a cut-based peephole resynthesis pass that restructures logic to push computation across ownership boundaries, maximizing what can be done locally while preserving functionality. A decomposition-driven synthesis engine (based on *Ashenhurst–Curtis* style reasoning [10, 67]) supplies low-JMC replacements for highlighted cuts, and a global accept/reject policy commits only those rewrites that strictly decrease JMC in the full circuit.

Conceptually, this delivers an orthogonal and complementary dimension to Chapter 6: whereas Chapter 6 improves the *ciphertext efficiency* of nonlinearity via the X1G representation, Chapter 7 *shrinks the amount of nonlinearity that must be garbled in the first place*. Together, they reduce both the *cost per* jointly evaluated non-linear gate and the *number of* such gates, driving down GC communication end-to-end.

To aid quick orientation, Table 1.1 distills the five technical chapters into a single view, mapping each chapter to its target scheme, core problem, adopted logic representation, the optimization objective, and headline gains. Readers can use this table as a roadmap and then consult the corresponding chapters for details on formulations, proofs, algorithms, and full experimental results.

Chapter 8 (Conclusions and Outlook). This chapter closes the thesis in three parts. First, it summarizes the five technical chapters:

(i) leveled FHE: joint, Pareto-guided optimization of multiplicative depth and multiplicative complexity;

(ii) TFHE under a single plaintext space: PBS-efficient technology mapping that exploits larger LUTs nd MV-PBS amortization;

(iii) TFHE with mixed ciphertext spaces: an encoding–space switching framework that coordinates binary/large-space gate evaluation via PBS;

(iv) GC with a new representation: X1G and an algebraic XAG→X1G mapping with X1G-specific optimizations;

(v) GC with ownership awareness: node ownership, joint multiplicative complexity, and a peephole resynthesis flow that minimizes jointly evaluated nonlinearity.

Second, it distills cross-cutting lessons: representation choice directly enables savings and new optimizations (e.g., LUT vs. XAG in TFHE; X1G vs. XAG in GC), and cost models must

track cryptographic reality (e.g., MD-MC composite for leveled FHE; JMC for GC), not just traditional circuit proxies.

Third, it offers higher-level guidance for the community: versioned cost-model interfaces between libraries and compilers, reference benchmarks and workloads, and better-defined APIs to hardware backends. These practices may stimulate advances across cryptography, compilers, and hardware, thereby accelerating the path toward practical and widely deployable secure computation.

Part	Scheme	Principal Contribution	Logic Representation	Cost Model	Representative Gains
I (Ch. 3)	Leveled FHE	*Multiplicative depth* (MD) and *multiplicative complexity* (MC) joint optimization	*XOR-AND-inverter graph* (XAG)	$\mathrm{MC} \times \mathrm{MD}^2$	21.32% average reduction in evaluation latency
I (Ch. 4)	TFHE, single plaintext space	PBS-efficient TFHE circuit mapping with a large plaintext space and *multi-value PBS* (MV-PBS) awareness	*Look-up table* (LUT) network with a dedicated gate-set	# PBS with MV-PBS amortization	29.94% average reduction in evaluation latency
I (Ch. 5)	TFHE, dual/multi plaintext spaces	Encoding–space switching: plan when to compute in binary (cheap linear ops) vs. larger spaces (compact non-linear LUTs)	XAG	# PBS	23.34% average reduction in evaluation latency
II (Ch. 6)	GC	Ciphertext-efficient logic representation with dedicated optimization algorithms	*XOR–OneHot-inverter graph* (X1G)	# OneHot	Up-to 26.14% average reduction in communication cost
II (Ch. 7)	GC	Ownership-aware optimization that minimizes *joint computation*	XAG	#AND that requires joint computation	3.30% average reduction in communication cost

Table 1.1: Overview of the five technical chapters: secure-computation scheme, problem focus, logic representation, cost model, and representative gains. Part I targets FHE (leveled and fast-bootstrapping); Part II targets GC. Quantitative results are summarized here at a high level; full datasets appear in each chapter's experimental section.

2 Preliminaries

2.1 Logic Synthesis Preliminaries

Logic synthesis is the process of transforming a high-level description of a Boolean computation into an optimized gate-level representation. In its *narrow* sense, logic synthesis asks: given a Boolean function, produce a high-quality implementation in a specified target representation, e.g., *exclusive-or sum of product* (ESOP) [158], *binary decision diagram* (BDD) [38], *AND-inverter graph* (AIG) [127], *look-up table* (LUT) network [62]. "High-quality" is application dependent and must be judged with respect to a *cost metric* defined on that representation.

Representation and cost are application-driven. Different application domains naturally privilege different forms and objectives. In secure computation, for instance, *XOR-AND-inverter graphs* (XAGs) are attractive because they separate linear (XOR) from non-linear (AND) primitives — mirroring the common asymmetry in cryptographic cost models where XORs are cheap while ANDs dominate either computation (FHE) or communication (GC). In such settings, the primary objective is often to *minimize the number of non-linear nodes*. Other examples abound: for *exclusive-or sum of prodcut* (ESOP) forms, one may minimize the number of product terms and/or bound cube arity [144]; for *binary decision diagrams* (BDDs), one typically targets finding a variable ordering such that node count [188], sometimes complemented by depth [109], can be minimized; for netlists mapped to a standard cell library, the metric may be total cell area or critical-path delay [169].

We briefly review the two ingredients we will rely on throughout the thesis:

(i) *logic representations* that expose structure relevant to our cost models, and

(ii) *logic optimization techniques* that reshape a network toward a chosen objective.

The review is not exhaustive, but focuses on the machinery that directly underpins the contributions of this thesis.

2.1.1 Logic Representation

The same Boolean function admits many semantically equivalent, yet operationally distinct, representations — ranging from truth tables, two-level representations (e.g., sum of products, exclusive-or sum of products), and multi-level representations (e.g., look-up table networks, AND-invertor graphs). No single representation is universally "best." Each exposes a different structure (algebraic, topological, or combinational), offers different algorithmic affordances (canonicality, algebraic rewriting rules, cut-based logic rewriting, etc.), and aligns with different cost models. Consequently, the most suitable choice is *use dependent*: what is ideal for mapping into garbled circuits may be suboptimal for leveled or fast-bootstrapping FHE, and vice versa.

This section briefly introduces the specific representations employed later in the thesis. The discussion is intentionally selective rather than exhaustive; our goal is to equip the reader with just enough background to understand the optimization problems and algorithms that follow, not to survey the full landscape of Boolean representations.

Truth Table

A truth table lists the output of a single-output function $f : \{0,1\}^n \mapsto \{0,1\}$, for every input assignment. With a fixed variable order and input enumeration, it is canonical: each function corresponds to exactly one bitstring.

Notation in this thesis. We encode a single-output n-input function as a 2^n-bit string prefixed by $^\#$, with the rightmost bit equal to $f(0,\ldots,0)$ (LSB). For compactness, we sometimes use hexadecimal (four bits per digit) as well. Examples: the truth table of the three-input majority function is $^\#11101000(^\#e8)$; for the two-input XOR function, it is $^\#0110(^\#6)$.

Use and limits. Truth tables are ideal for exact specification, small-cut matching, cofactoring, and property checks. However, they scale poorly — space/time $\mathcal{O}(2^n)$ — so, we reserve them for small subproblems and rely on other forms for large functions.

Exclusive-or Sum of Products (ESOP) Form

An ESOP is a two-level, polarity-aware form that writes $f : \{0,1\}^n \mapsto \{0,1\}$ as an XOR of product terms (cubes):

$$f(x_1,\ldots,x_n) = P_1 \oplus P_2 \oplus \cdots \oplus P_m, \quad P_i = \bigwedge_{x_j \in S_i} \ell_{ij},$$

where each literal $\ell_{ij} \in \{x_j, \neg x_j\}$ and variables not in S_i are don't-cares for cube P_i. ESOPs are not canonical — many distinct ESOPs may compute the same function.

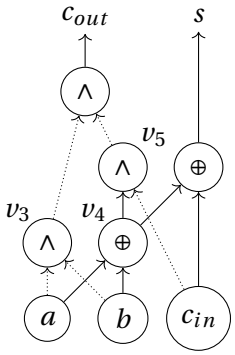


Figure 2.1: An XAG implementation of a one-bit full adder.

Minimizing ESOP cost (e.g., number of cubes, total literal count) is NP-hard, but there exist powerful heuristics and exact methods [144, 129, 153]. Because all non-linear ANDs appear in one level and combine through XOR, ESOPs are a strong fit when XOR is cheap/free and the dominant objective is to limit non-linear depth — e.g., in cryptographic and quantum settings focused on multiplicative-depth reduction [95, 192].

Logic Network

Basic concepts. A logic network, Boolean circuit, is a *directed acyclic graph* (DAG) $G = (V, E)$ whose vertices V are either *primary inputs* (PIs) or logic gates, and whose directed edges E model wires. An edge $n_i \to n_o \in E$ indicates that n_i is a *fan-in* of n_o and n_o is a *fan-out* of n_i. Edge-level complementation (inverters) is recorded as an attribute, keeping the structure compact. The set of PIs is $I \subseteq V$; the set of *primary outputs* (POs), O, consists of references (optionally complemented) to nodes in V.

Why multi-level networks. We adopt multi-level logic networks as a primary representation for three reasons:

- **Scalability.** Multi-level graphs scale to large functions; by contrast, other forms, such as truth tables and ESOP, grow exponentially and become intractable beyond a few dozen inputs.
- **Optimization leverage.** The graph structure admits fast, local transformations.
- **Cost fidelity.** There is a near-structural isomorphism between a logic network and the target circuit realization. Thus, optimizing the network is a reliable proxy for the eventual circuit implementation.

XOR-AND-inverter graphs (XAGs). Restricting the gate library to {XOR, AND, NOT} yields an *XOR-AND-inverter graph* (XAG). NOT is modeled as edge complementation — dotted

edge indicates logic complementation; internal nodes are typically two-input XORs ($\oplus$) and two-input ANDs ($\wedge$). Figure 2.1 shows an XAG for a one-bit full adder. Throughout the thesis, we choose the node library to match the target secure-computation scheme.

Multiplicative complexity (functional vs. structural). The *multiplicative complexity* (MC) of a Boolean function is a measure of its intrinsic nonlinearity: it is the minimum number of AND gates required to realize the function over the basis{XOR, AND, NOT} [34]. This is a *functional* quantity — it depends only on the Boolean function itself (equivalently, its truth table), not on any particular implementation.

In contrast, when working with a concrete XAG we will often speak of *structural MC*, namely the number of AND nodes *present in that implementation*. Structural MC is the direct driver of cost in several settings; for example, in the task of garbled circuit generation, it coincides with the number of non-linear gates that must be garbled and communicated [193], and thus has served as the principal optimization target in GC-oriented synthesis [152, 172, 171]. In the XAG of Figure 2.1, the structural MC equals two because there are exactly two AND nodes.

By definition, the functional MC of a function lower-bounds the structural MC of *any* of its XAG realizations. Computing functional MC is challenging in general: beyond special classes (e.g., symmetric functions [33]), the best exact methods determine MC only for small input sizes (up to about six variables) [39]. Consequently, practical flows rely on heuristic or guided synthesis to minimize structural MC — that is, to search for XAGs with as few AND gates as possible — even when the true functional MC is unknown [171].

Cuts. Many effective, heuristic logic optimization techniques follow a *peephole* strategy: expose overlapping sub-networks and improve them locally. Two structural notions are central: *cuts* and the *maximum fanout-free cone* (MFFC). A cut C highlights a logic cone via a root r and a set of leaves L, such that

(i) every PI-to-r path passes through at least one $l \in L$, and

(ii) each $l \in L$ lies on some PI-to-r path.

A cut is k-feasible if $|L| \leq k$. In Figure 2.1, three nontrivial 3-feasible cuts rooted at c_{out} are $C_1 = \{v_3, v_5\}$, $C_2 = \{v_3, v_4, c_{in}\}$, and $C_3 = \{a, b, c_{in}\}$. *Cut enumeration* [63] lists all k-feasible cuts for each node, exposing many *local views*.

Let $f_n : \mathbb{B}^{|I|} \mapsto \mathbb{B}$ be the *global function* at node n. Given a cut C for n with leaves L, the corresponding *local function* is $f_n^C : \mathbb{B}^{|L|} \mapsto \mathbb{B}$, i.e., f_n seen in terms of L only. In Figure 2.1, $f_{c_{out}}^{C_1} = \neg v_3 \wedge \neg v_5$. Since C_3 spans the PIs, $f_{c_{out}}^{C_3} = f_{c_{out}} = \langle a, b, c_{in} \rangle$ (majority). Dynamic programming over cuts selects locally improved cones to minimize a global cost [63, 128].

Maximum fanout-free cone (MFFC). The MFFC of node n is the set of nodes in n's transitive fan-in such that every path from any of those nodes to *any* PO passes through n. Intuitively, nodes in MFFC(n) are used *exclusively* by n; removing n lets us delete its MFFC without affecting other outputs. For example, in Figure 2.1, the MFFC of node c_{out} consists of node v_3 and v_5. Conversely, there are no nodes in the MFFC of node s, as its two fan-in nodes, node v_4 and node c_{in}, also contribute to node v_5. MFFC size estimates a local "area" contribution and underlies popular accounting heuristics such as *area flow* (sharing-aware) [121] and *exact area* (MFFC-based) [130].

2.1.2 Logic Optimization

From one-shot synthesis to flows. Producing an optimal implementation in one step is rarely feasible: across representations and objectives, the underlying optimization problems are typically NP-hard [138]. Exact approaches scale poorly (see the sub-subsection on "Exact Synthesis"). Hence, practical toolchains adopt a *flow*:

(i) *Initial synthesis* constructs a workable implementation in the target (or an *intermediate*) representation — common choices include AIGs due to their simplicity and amenability to fast transformations;

(ii) *Logic optimization* applies heuristics to improve the implementation w.r.t. the cost metric;

(iii) *Technology mapping* (when an intermediate form is used) converts the optimized network into the final target while attempting to preserve, or even increase, the gains.

Consequently, in this thesis, the term "logic synthesis" generally refers to the *end-to-end process*, and our contributions predominantly focus on the *optimization* stage (including mapping when it is cost-aware).

Below, we review core techniques frequently employed in logic optimization.

Exact Synthesis

Exact synthesis seeks a *provably optimum* realization of a Boolean function in a fixed representation under a fixed cost metric. Modern approaches commonly encode the search as *Boolean satisfiability* (SAT), leveraging powerful SAT solvers [164]. The quality of the SAT formulation strongly influences scalability [91]. Despite steady progress, exact synthesis remains intractable in the worst case [138] and is therefore practical mainly for small functions or highly structured subproblems. In practice, exact methods are often used *locally* (e.g., inside sub-circuits) or to pre-compute optimal implementations for small functions that are then reused during heuristic circuit rewriting.

Boolean Function Classification

Classifying Boolean functions up to natural equivalences is essential because the design space explodes doubly exponentially: there are 2^{2^n} n-input Boolean functions. This growth makes many otherwise-straightforward tasks impractical — for example, building a complete database of optimal implementations (to accelerate logic rewriting or technology mapping) becomes infeasible if every truth table is treated as a distinct entry. By categorizing functions into equivalence classes and choosing a canonical representative per class, we shrink storage by orders of magnitude, make table lookups fast, and allow any synthesis result computed for a representative to be reused for all members of the class. We briefly review two notions relevant to this thesis below.

NPN classification. Two n-variable functions are *NPN-equivalent* if one can be obtained from the other by input negation, input permutation, and output negation [87]. Storing one representative per NPN class drastically compresses libraries used in rewriting.

Affine/spectral classification. Affine (or spectral) equivalence extends NPN with linear transformations over $\mathbb{B}$: variable translations $x_i \mapsto x_i \oplus x_j$ and global translations $f \mapsto f \oplus x_i$ [77]. These operations preserve key algebraic properties and are useful in cryptographic settings.

Crucially, the choice of equivalence relation is *application driven*: two functions belong to the same class precisely when the transformations that map one to the other are (nearly) free in the target setting. In CMOS-oriented digital design, NPN classification is natural because input/output inversions and wire permutations are inexpensive relative to other gates. In cryptographic analysis, affine classification is preferred since affine changes of variables preserve algebraic degree, nonlinearity, and spectral characteristics — properties central to security metrics — while linear layers are typically modeled as costless or low-cost.

Algebraic (Rule-Based) Rewriting

Algebraic methods treat logic networks as algebraic objects and apply fast, local axioms to reshape the graph. Because these transformations ignore some Boolean constraints, they are extremely efficient and scale to large designs. Typical passes include factoring/distribution, common-sub-expression extraction, re-association, and depth/area balancing. Such methods are especially potent on near-homogeneous networks (e.g., AIGs, *majority-inverter graphs* (MIGs) [4]) where local patterns recur and can be recognized and applied cheaply.

Logic Rewriting

Logic rewriting optimizes a network by replacing small, selected sub-circuits with (near-)optimal implementations of their Boolean functions. In contrast to algebraic (rule-based)

rewriting, which applies syntax-level identities within a chosen algebra, logic rewriting is *semantics-aware*: it explicitly reasons about Boolean equivalence of the highlighted region and its replacement. This Boolean awareness enables more powerful transformations that may change structure and primitive types while preserving function.

Workflow. A typical pass proceeds as follows.

1. **Region selection.** Enumerate k-feasible cuts for each node. Each cut $C = (r, L)$ induces a sub-network and a local Boolean function $f_C : \mathbb{B}^{|L|} \mapsto \mathbb{B}$ at the root r.
2. **Function extraction & canonicalization.** Compute a compact representation of f_C (e.g., truth table or AIG hash), optionally canonicalize it under an application-appropriate equivalence (NPN or affine; cf. "Boolean Function Classification") to enable database lookup and result reuse.
3. **Candidate generation.** Obtain a high-quality implementation for f_C in the *target representation*:
   - *On-the-fly synthesis* via a dedicated engine (exact or heuristic).
   - *Database lookup* of precomputed optimal/near-optimal implementations for the canonical representative.
4. **Gain estimation.** Evaluate the global impact of replacing the current realization of f_C by the candidate.
5. **Commit or skip.** If the estimated cost (area, depth, non-linear count, etc.) improves and constraints are met (e.g., depth bound), apply the replacement; otherwise discard. Iterate in topological passes until convergence.

Synthesis engines: exact vs. heuristic. The design of the local synthesis engine is technically central: it must strike a balance between *quality* (how close to optimal under the chosen cost metric) and efficiency.

- **Exact engines** (SAT- or *integer linear programming* (ILP)-based) provide provably optimal implementations for small k or for constrained objectives (e.g., depth-optimal AIGs [6]). Their scalability is limited, so they are typically used with small cuts or selectively on hotspots.
- **Heuristic engines** (pattern growth, Boolean decomposition, etc.) generate high-quality candidates quickly and scale to larger k (e.g., [189]).

Importantly, *local optimality does not guarantee global optimality*: a replacement that is optimal for f_C may reduce logic sharing or increase downstream cost.

The design of these local synthesis engines is a recurring theme in this thesis. We will revisit it with different objectives and representations: an exact engine tailored to leveled-FHE cost models in Chapter 3, an ESOP-guided cut-replacement engine for encoding-switch-aware TFHE in Chapter 5, an *XOR-OneHot-inverter graph* (X1G)-oriented exact engine in Chapter 6, and an *Ashenhurst–Curtis decomposition* (ACD)-based engine for low-garbling-cost GC generation in Chapter 7. Each instance illustrates how tuning the engine to the application-specific representation and metric can be as impactful as the surrounding optimization framework.

Don't-Care–Based Optimization

Don't cares (DCs) are input conditions for which the output is unspecified. They arise structurally from reconvergent fanout and environment constraints and are classically divided into:

(i) *Controllability don't cares* (CDCs): minterms that never occur at a node due to upstream structure;

(ii) *Observability don't cares* (ODCs): output variations that are unobservable at any PO due to downstream masking [36].

Exploiting DCs enlarges the space of admissible local replacements and can yield substantial reductions. Modern flows compute and localize DC information via SAT solving [126], and then drive optimization using this flexibility.

2.2 Cryptographic Preliminaries

2.2.1 Fully Homomorphic Encryption

This subsection gives a concise, self-contained introduction to *fully homomorphic encryption* (FHE), focusing on shared concepts across schemes.

Homomorphism, noise, and bootstrapping. An FHE scheme enables computation over encrypted data by providing ciphertext-domain operations that track their plaintext counterparts. Informally, if Enc and Dec are the encryption and decryption algorithms, a homomorphic addition/multiplication satisfies

$$\mathsf{Dec}\big(\mathsf{Enc}(x) \oplus \mathsf{Enc}(y)\big) \approx x + y$$
$$\mathsf{Dec}\big(\mathsf{Enc}(x) \otimes \mathsf{Enc}(y)\big) \approx x \cdot y$$

where $+/\cdot$ and $\oplus/\otimes$ are addition/multiplication in the plaintext and ciphertext domains, respectively; $\approx$ means equality for exact schemes (BGV/BFV) and a controlled approximation

for approximate schemes (CKKS). Supporting both addition and multiplication suffices to realize arbitrary arithmetic circuits over the message space; with a binary message space setting, addition and multiplication correspond to Boolean XOR and AND operations, thus covering Boolean circuits as well.

While the encryption depends on a specific scheme, ciphertexts typically carry a *noise* term, as the security comes from the underlying (*ring*) *learning-with-error* ((R)LWE) problem [149]. The noise component is benign at creation but *grows* under homomorphic operations. Additions increase noise mildly; multiplications inflate it significantly. If noise exceeds a scheme-dependent threshold, decryption fails. Thus, *multiplicative depth* (MD) — the length of the longest multiplication chain — limits how far one can evaluate before noise must be reduced.

Noise can be *reset* by *bootstrapping*, i.e., homomorphically evaluating a low-depth decryption circuit using an encrypted secret key so that the refreshed ciphertext can sustain further computation. Leveled schemes choose parameters large enough to finish a computation without bootstrapping; fast-bootstrapping schemes pay frequent, cheap refreshes instead.

Rings, encodings, and packing. Practical schemes operate over polynomial structures: a plaintext ring (for user data) and a ciphertext ring (for encryptions) defined modulo a cyclotomic polynomial. Many schemes exploit *SIMD packing*: via the *Chinese Remainder Theorem* (CRT), a plaintext polynomial can encode dozens to thousands of *independent slots*, enabling vector-style parallelism; *automorphisms* implement *slot rotations* homomorphically to realize data movement among slots. These features are crucial for throughput, but rotations typically require key switching and are therefore relatively costly compared to slot-wise addition and multiplication.

Security posture. Security reductions rely on post-quantum hardness ((R)LWE-type assumptions) with dedicated concrete estimates [2] and standardized parameter choices [1]. It is worth noting that baseline FHE provides confidentiality, but not integrity or circuit privacy, unless deliberately augmented [59, 120, 105].

2.2.2 Modern FHE Schemes

This subsection provides two mainstream instantiations used throughout this thesis: BFV [81], representing leveled schemes, and TFHE [57], representing fast-bootstrapping schemes.

BFV

Design goal. BFV [81] is a ring-LWE-based, leveled scheme: choose parameters large enough to evaluate a target circuit without invoking bootstrapping, then leverage SIMD packing and automorphisms for improving throughput.

Spaces and encoding. BFV targets exact modular arithmetic over $\mathbb{Z}_p$, where p is the plaintext space size. Plaintexts are polynomials of degree $< n$ with coefficients in $\mathbb{Z}_p$; ciphertexts are tuples of polynomials in the analogous ring modulo a larger modulus q, also referred to as the ciphertext space size. While a scalar message $m \in \mathbb{Z}_p$ can be naïvely embedded as a constant polynomial, a common practice is that one reinterprets the plaintext ring — via CRT — as many independent *slots* so that a single plaintext encodes a vector (often thousands of integers). In this way, homomorphic addition and multiplication over ciphertexts act *slot-wise*, achieving the effect of SIMD execution. Meanwhile, automorphisms enable *cyclic rotations* of these slots for data movement.

Operations and maintenance. Each ciphertext has two components, namely, $[c_0, c_1] \in R_q^2$. Multiplying two ciphertexts yields a three-component ciphertext (quadratic in the secret). To keep ciphertexts linear in the secret (and to control noise), BFV applies *relinearization*, a key-switch–based procedure that collapses a degree-2 ciphertext back to degree-1. BFV is typically used without bootstrapping (leveled mode). Therefore, synthesizing a low-MD circuit implementation allows choosing a modest budget, and careful planning of relinearization and modulus switching is essential to finish the computation within the chosen budget.

Performance profile. BFV's strengths are *throughput* (thanks to large SIMD batches) and exact modular arithmetic. The main homomorphic evaluation costs are ciphertext multiplications, slot rotations, and the memory traffic due to large RNS moduli. This makes BFV attractive for data-parallel workloads (e.g., vector arithmetic) where the circuit's MD is modest and data movement is structured.

TFHE

Design goal. TFHE exemplifies the FHE family that prioritizes *fast bootstrapping* over batching. Instead of sizing parameters to avoid noise refresh, TFHE embraces frequent refresh and uses each bootstrap to *compute* a small function (a *gate* or *LUT*) while resetting noise. This yields sub-100ms bootstraps in optimized software [57], which can be further reduced to as low as 35μs on dedicated hardware accelerators [175], enabling low-latency computation over binary values or small integers.

Spaces and ciphertext types. TFHE's ciphertexts live over the *torus*, $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ (i.e., reals modulo 1), organized as polynomials modulo an irreducible polynomial. This domain natively supports linear operations (notably addition and scalar multiplication), but ciphertext–ciphertext multiplication is not a primitive. To accommodate this, nonlinearity is introduced via bootstrapping using *GSW*-style ciphertexts [83] that encrypt (gadgeted) secret-key material: during bootstrapping, torus ciphertexts are transformed by multiplying them with these encrypted keys. In the commonly implemented *gate-bootstrapped* variant, Boolean gates and small LUTs

are realized by *programmable bootstrapping* (PBS). We defer the detailed construction of PBS to Chapter 4, where it is presented alongside our proposed improvements.

Programming model and primitives. The scheme offers efficient MUX gates and general LUT evaluation inside the bootstrap, so circuits can be described as networks of LUTs with small fan-in (e.g., two-input logic gates as described by the `TFHE` library). More advanced constructions (e.g., weighted finite automata) expand the design space for arithmetic functions, facilitating advanced applications such as private-preserving DNN inference [54]. In this scheme, since bootstrapping plays both the role of proceeding computation and resetting noise, performance is largely driven by the number of PBS operations, while per-PBS latency largely depends on the gate arity.

Summary. In summary, TFHE forgoes SIMD batching: one gains low latency per bootstrapping but loses massive slot parallelism. As a result, TFHE excels for latency-sensitive, control logic-heavy workloads, while high-throughput vector workloads may favor leveled schemes with batching [176].

2.2.3 Garbled Circuits

The GC Protocol

Originally proposed by Andrew Yao for *two-party computation* (2PC) [190] and later generalized to MPC [18], the *garbled circuit* (GC) protocol evaluates a Boolean circuit on private inputs without revealing intermediate values. We sketch the 2PC version (the core of MPC variants), following the formalization style of [19].

In the protocol, two parties—commonly referred to as the *garbler* and the *evaluator* — jointly compute a Boolean function

$$f : \{0,1\}^n \mapsto \{0,1\}^m$$

over their private inputs without revealing them to each other. The function f is represented as a Boolean circuit composed of logic gates.

Wire labels and garbled tables. At a high level, the garbler assigns two random wire labels $W_0, W_1 \in \{0,1\}^k$ to each wire w in the Boolean circuit, where k is the security parameter. These labels encode the semantic values 0 and 1, respectively. For each gate g with input wires u and v (assuming gate g has two inputs) and output wire w, the garbler constructs a *garbled truth table* consisting of four ciphertexts. For each input assignment $(a,b) \in \{0,1\}^2$, the garbler computes a ciphertext that encrypts the output label W_c, where $c = g(a,b)$, under the keys U_a and V_b. This is typically realized using symmetric-key encryption as in the original

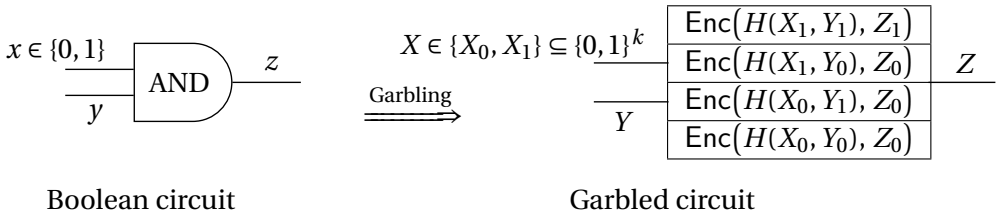


Figure 2.2: From a plain AND gate to its four-row garbled truth table.

construction [190] or, equivalently, via a pseudo-random function [19]. Garbling all gates in this manner yields a *garbled circuit*[1] for f, which the garbler transmits to the evaluator.

A concrete example. Figure 2.2 depicts this transformation for a two-input AND gate. On the left is the plain Boolean gate with inputs $x, y \in \{0,1\}$ and output $z = x \wedge y$. On the right is its garbled form: a four-row table whose entries are ciphertexts of the appropriate output label Z_c (here $c = a \wedge b$) under the corresponding input labels X_a and Y_b. The horizontal arrow labeled "garbling" highlights the one-to-one conversion from gate to table, where $X \in \{X_0, X_1\} \subseteq \{0,1\}^k$, $Y \in \{Y_0, Y_1\}$, and $Z \in \{Z_0, Z_1\}$ denote the wire labels for the inputs and output. In the full garbling of a circuit, this conversion is performed gate-by-gate; the resulting network of tables is exactly isomorphic to the original Boolean circuit.

Input acquisition and evaluation. The evaluator obtains the wire labels corresponding to its private input via an *oblivious transfer* protocol and receives the labels for the garbler's input directly from the garbler. For any gate g during evaluation, the evaluator therefore holds exactly one label per input wire, say U_a and V_b. In tandem with the *point-and-permute* technique [18], the evaluator always knows which row of the table to attempt to decrypt. Using these as decryption keys, the evaluator decrypts *exactly one* ciphertext in the gate's garbled truth table — recovering the output label $W_{g(a,b)}$. That output label then feeds subsequent gates as their input label, and the process repeats in topological order. Throughout, the evaluator manipulates only opaque labels and thus learns nothing about intermediate or private values beyond the final output.

Output decoding. After evaluating the last gate, the evaluator holds a garbled output label for each primary output wire. The garbler provides the corresponding decoding information, enabling the evaluator to map each output label back to its plaintext bit and thereby obtain f's cleartext output.

[1] Throughout this thesis, we use "GC" when referring to the secure computation protocol based on garbled circuits, and reserve the full term "garbled circuit" for the cryptographic encoding of a Boolean circuit.

System Performance Bottleneck in Garbled Circuit Execution

A GC is a network of encrypted tables; each table entry is a *ciphertext* that must be communicated to the evaluator. The *garbling cost* — the total number of ciphertexts transmitted (equivalently, the total table size) — dominates overall latency and bandwidth. Large applications can be communication-bound; e.g., garbling a modest three-layer neural network already entails on the order of hundreds of megabytes of data [17].

Because the Boolean circuit and its GC are essentially isomorphic (each logic gate ↔ one encrypted table), there are two orthogonal avenues to reduce garbling cost:

1. **Logic-level (representation) optimizations:** choose and optimize the circuit representation to favor gates with low ciphertext footprint under modern garbling schemes.

2. **Scheme-level optimizations:** improve how individual gates are garbled to shrink per-gate table sizes.

2.2.4 Advanced Garbling Techniques

Within the two orthogonal axes that we approach reducing the garbling cost of secure computation along, the contributions in Chapters 6–7 belong to the logic-level-optimization axis: we propose logic representations and synthesis flows that restructure circuits to be more garbling-efficient. Crucially, these logic-level methods are designed to be *practical* — they assume (and fully exploit) the strongest scheme-level constructions available today. In particular, we adopt *free-XOR* [108] for linear gates, *garbled row reduction (GRR3)* [139] and *half-gates* [197] for nonlinear gates, and, where applicable, *garbling gadgets* for larger symmetric gates [16]. For completeness, and to make clear the baseline our results build upon, we briefly review these techniques below. Because garbling gadgets are tightly coupled to our reasoning and to the logic representation we introduce in Chapter 6, we defer the relevant discussion to the preliminary section of that chapter (Section 6.2.1).

Free-XOR Optimization

The *free-XOR* technique [108] makes XOR gates in a garbled circuit essentially free — no ciphertexts to generate, transmit, or decrypt. The garbler samples a secret *global offset* $\Delta \in \{0,1\}^k$ (kept fixed for the whole circuit) and encodes each wire w using a pair of k-bit labels

$$W_1 = W_0 \oplus \Delta,$$

where $\oplus$ is bitwise XOR. With this affine structure, the evaluator can compute the output label of an XOR gate with a single label XOR:

$$W_{a\oplus b} = U_a \oplus V_b.$$

Therefore, no garbled table is needed for XOR.

Efficient Garbling of Nonlinear Gates

Nonlinear gates (e.g., AND) dominate GC cost because they require ciphertexts. Two widely used optimizations — both compatible with free-XOR — are *garbled row reduction* (GRR3) and *half-gates*.

Garbled Row Reduction (GRR3). Yao's classical garbling uses four ciphertexts for a two-input gate. GRR3 [139] reduces that to three by fixing one row (typically the $(0,0)$ row) to an all-zero string that need not be transmitted. Concretely, for a gate g with inputs u, v and output w, ciphertexts are of the form

$$C_{a,b} = \mathsf{Enc}\big(H(U_a, V_b),\, W_{g(a,b)}\big), \quad a, b \in \{0,1\},$$

where H is a key-derivation/hash function and Enc is a symmetric cipher or XOR-pad. The garbler chooses $W_{g(0,0)}$ so that $C_{0,0} = \{0\}^k$, namely, a string of k zero-s. Using point-and-permute [18], the evaluator identifies the correct row and never needs the omitted ciphertext — saving one transmission per gate.

Half-Gates. Building on GRR3, the *half-gates* construction [197] achieves two ciphertexts per two-input AND, which is optimal under standard assumptions. It uses the identity

$$a \wedge b = (a \wedge r) \oplus (a \wedge (b \oplus r)),$$

for any bit r. The garbler sets r (e.g., to the select bit of W_0) and reveals $p = b \oplus r$ to the evaluator via the label's select bit, which leaks nothing about b. The AND is split into:

- *Garbler half-gate*: computes $a \wedge r$. Since r is known to the garbler, the computation is garbled with *one* ciphertext (via GRR3).
- *Evaluator half-gate*: computes $a \wedge (b \oplus r)$. Because the evaluator knows $p = b \oplus r$, the computation is also garbled with *one* ciphertext (again via GRR3).

XORing the two half-gate outputs yields the correct output label for $a \wedge b$. Combined with free-XOR for all linear wiring around ANDs, the *half-gates + free-XOR* toolkit is widely adopted as the baseline for communication- and compute-efficient GC protocols.

2.3 Benchmark Suites

To assess the performance of the algorithms proposed throughout this thesis, we employ a diverse set of benchmark suites, each designed to capture different aspects of secure Boolean

computation. Rather than introducing these suites separately in each technical chapter, we consolidate their descriptions here to avoid redundancy and to provide a unified reference point.

Each benchmark suite is introduced with a brief overview of its origin, the types of circuits it contains, and its relevance to secure computation. All benchmark circuits are first mapped into the XAG format, which — as discussed in Section 2.1.1 — is a logic representation of particular interest in cryptographic contexts due to its alignment with the cost models of many secure computation protocols. For completeness, we also summarize the characteristics of each benchmark circuit in tabular form, including input/output sizes and multiplicative complexity (i.e., number of AND gates), which serve as key indicators of cryptographic cost under various protocols.

It is important to note that *not* all benchmark suites are used in every technical chapter. Since each suite is tailored toward a particular class of applications or security settings, we select benchmarks based on their alignment with the research objective of the corresponding chapter. For example, benchmarks targeting SMPC are not considered in chapters focused on FHE (Part I), as the latter targets SOC tasks. Additionally, we prioritize suites for which baseline or state-of-the-art performance results are available, to enable meaningful comparisons. For instance, the EPFL benchmark suite, which is general-purpose, is used in our GC evaluations (Part II) to align with prior work on optimizing garbling cost via MC reduction [152, 172, 171, 118].

All benchmarks considered in this thesis represent *combinational Boolean functions*, consistent with our focus on optimizing secure computation over Boolean logic.

2.3.1 EPFL Combinational Benchmark Suite

The EPFL Combinational Benchmark Suite was proposed to serve as a comparative standard for logic synthesis and optimization tools [5]. It comprises originally 23 purely combinational circuits, partitioned into three categories: arithmetic, random/control, and "MtM" (more-than-million) circuits. We exclude the three MtM benchmark circuits — as aligned with prior works' setting [172, 171, 118] — given their extreme size and intractability for many secure-computation schemes.

Although the original benchmarks are provided in AIG format, for this thesis, we use their optimized, known MC-minimal XAG versions, jointly collected from prior works ([152, 172, 171, 118]). Their profiles (after XAG conversion and optimization) are summarized in Table 2.1.

Benchmark	#Inputs	#Outputs	MC
Arithmetic			
adder	256	129	128
barrel shifter	135	128	832
divider	128	128	5 132
log2	32	32	8 773
max	512	130	872
multiplier	128	128	7 585
sine	24	25	1 959
square-root	128	64	5 217
square	64	128	4 596
Random/Control			
round-robin arbiter	256	129	1 174
coding-cavlc	10	11	394
ALU control unit	7	26	45
decoder	8	256	328
i2c controller	147	142	557
int to float converter	11	7	85
memory controller	1 204	1 231	4 695
priority encoder	128	8	323
look-ahead XY router	60	30	93
voter	1 001	1	4 257

Table 2.1: Profile of the EPFL benchmark suite after MC-minimal XAG conversion.

2.3.2 Cryptographic Benchmark Suite

The Cryptographic Benchmark Suite is a subset of the publicly available *Bristol-style MPC circuits*[2] originally dedicated to support SMPC evaluations. It includes hand-crafted, gate-level implementations of various cryptographic primitives and comparison circuits in a uniform, textual format compatible with many SMPC frameworks.

For this thesis, we adopt a subset of these circuits focused on symmetric cryptography and integer comparison, following the evaluation setup used in prior works to ensure fair comparison. Specifically, we include:

- **AES**: both with and without key expansion,
- **DES**: both with and without key expansion,
- **Integer comparison**: four versions of 32-bit comparisons, covering signed/unsigned and strict/non-strict variants (namely, $<$ and $\leq$),
- **Hash functions**: MD5, SHA-1, and SHA-256.

[2]Available at: https://nigelsmart.github.io/MPC-Circuits/

Benchmark	#Inputs	#Outputs	MC
AES (KeyExp)	1 536	128	5 440
AES (No KeyExp)	256	128	6 800
DES (KeyExp)	832	64	6 915
DES (No KeyExp)	128	64	6 833
Comp. 32-bit SLT	64	1	84
Comp. 32-bit SLTEQ	64	1	87
Comp. 32-bit ULT	64	1	84
Comp. 32-bit ULTEQ	64	1	87
MD5	512	128	9 367
SHA-1	512	160	11 483
SHA-256	521	256	26 464

Table 2.2: Profile of the cryptographic benchmark suite after MC-minimal XAG conversion.

To reflect state-of-the-art circuit structures, we use the optimized versions of these circuits as presented in the works by Testa et al. ([172, 171]) and Liu et al. ([118]), which apply logic optimizations targeting low MC. Their profiles, including input/output sizes and number of AND gates, are summarized in Table 2.2.

2.3.3 MPCircuit Benchmark Suite

The MPCircuits Benchmark Suite is a parameterized collection of Boolean circuits for evaluating SMPC protocols [152]. The circuits target a range of privacy-sensitive, real-world applications, covering domains such as *auctions*, *statistical analysis*, *voting*, and *stable matching*. Unlike fixed-structure benchmarks, the `MPCircuits` implementation provides parameterized templates, allowing users to instantiate a variety of use cases and scales within each application class.

In this thesis, we adopt a representative subset of four circuit families — *auction*, *k-nearest-neighbor* (k-NN) search, *voting*, and *stable matching* — following the same parameter settings used in prior work [171] to ensure fair comparisons. The benchmark instances are summarized below:

- Auction circuits (`auction_n_b`): Each circuit receives 2^n bids of b bits each and returns the maximum bid and the ID of the bidder. We evaluate six parameter pairs: $(n,b) \in (2,16),(2,32),(3,16),(3,32),(4,16),(4,32)$.
- k-NN search circuits (`knn_comb_k_n`): Given a 32-bit reference value and n candidate values (also 32-bit), the circuit outputs the k values closest to the reference. We adopt: $(k,n) \in (1,8),(1,16),(2,8),(2,16),(3,8),(3,16)$.
- Voting circuits (`voting_n_m`): Each of the 2^m voters casts a vote for one of 2^n candidates

Benchmark	#Inputs	#Outputs	MC
auction_2_16	64	18	97
auction_2_32	128	34	193
auction_3_16	128	19	232
auction_3_32	256	35	456
auction_4_16	256	20	495
auction_4_32	512	36	975
knn_comb_1_8	288	32	554
knn_comb_1_16	544	32	1 162
knn_comb_2_8	288	64	881
knn_comb_2_16	544	64	1 919
knn_comb_3_8	288	96	1 060
knn_comb_3_16	544	96	2 394
voting_1_3	8	1	7
voting_1_4	16	1	15
voting_2_2	8	2	21
voting_2_3	16	2	55
voting_2_4	32	2	104
voting_3_4	48	3	275
stable_matching_4_8	192	24	16 001
stable_matching_8_8	384	24	59 773

Table 2.3: Profile of the MPCircuit benchmark suite after MC-minimal XAG conversion.

(by submitting an n-bit ID). The circuit outputs the ID of the winning candidate. We evaluate: $(n,m) \in (1,3),(1,4),(2,2),(2,3),(2,4),(3,4)$.

- Stable matching circuits (`stable_matching_r_s`): Each party in two groups of s members provides a preference list over r candidates from the other group, and the circuit returns a stable matching. We consider: $(r,s) \in (4,8),(8,8)$.

Table 2.3 summarizes the structure of each instance.

2.3.4 LOBSTER Benchmark Suite

The LOBSTER Benchmark Suite comprises 25 Boolean circuits curated by Lee et al. [112] to represent computations that are likely to become recurring components in future FHE applications. The benchmark collection emphasizes diversity and practical relevance, spanning a wide range of functional domains that align with secure computation needs.

Specifically, the suite includes:

- *FHE-targeted algorithms* from application domains such as medical diagnosis, stream ciphers, pattern search, and sorting, originally gathered from the `Cingulata` benchmark

Benchmark	#Inputs	#Outputs	MC
cardio	112	4	109
dsort	48	48	708
msort	48	48	810
isort	48	48	810
bsort	48	48	810
osort	48	48	702
hd01	32	32	87
hd02	32	32	76
hd03	16	8	27
hd04	16	8	75
hd05	64	32	121
hd06	64	32	121
hd07	8	8	17
hd08	8	1	18
hd09	32	32	134
hd10	32	32	35
hd11	32	32	391
hd12	32	32	116
bar	135	128	3 141
cavlc	10	11	655
ctrl	7	26	107
dec	8	256	304
i2c	147	142	1 157
int2float	11	7	213
router	60	30	170

Table 2.4: Profile of the LOBSTER benchmark suite.

collection [42];

- *Oblivious sorting algorithms* from [48];
- *Bit-twiddling primitives* from the *Hacker's Delight* collection [184];
- Selected circuits from the EPFL benchmark suite [5], included to provide coverage of general-purpose logic blocks.

Besides their potential as building blocks for FHE workloads, these circuits were selected since their structural diversity provides a strong foundation for stress-testing synthesis and optimization techniques under cryptography-aware cost models. The structural characteristics of each circuit are summarized in Table 2.4.

Part I

Efficient Fully Homomorphic Encryption from the Ground Up: Boolean-Level Techniques for Leveled and Torus FHE

3 Beyond Depth: Multiplicative Depth vs. Complexity in Leveled FHE

3.1 Motivation

As reviewed in Chapter 2, *fully homomorphic encryption* (FHE) enables computation directly over encrypted data [155], with Gentry's bootstrapping theorem establishing the first construction capable of arbitrary computation [82]. Despite substantial advances, homomorphic evaluation remains orders of magnitude slower than plaintext computation. Nevertheless, the ability to delegate computation without exposing plaintext makes FHE a compelling foundation for privacy-preserving applications, from outsourcing medical analytics [44] to private neural network inference [69]. Continued acceleration efforts are therefore essential.

Modern FHE broadly divides into *leveled schemes* (e.g., BGV [35], BFV [81]) and *fast-bootstrapping schemes* (e.g., FHEW [75], TFHE [57]). These families rely on fundamentally different constructions and thus invite different optimization objectives and techniques. This chapter focuses on *leveled* schemes and, specifically, on logic-level optimization of Boolean circuits to accelerate homomorphic evaluation under BFV/BGV, which support exact computation over encrypted bits.

Leveled schemes operate over large plaintext/ciphertext moduli and control noise growth through mechanisms such as modulus switching. With a sufficient noise budget — i.e., a ciphertext modulus composed of enough prime factors — a circuit can be evaluated without intermediate bootstrapping [134]. For functional completeness, an FHE scheme must support both additions and multiplications; in practice, homomorphic multiplication is significantly more expensive than addition for two reasons:

- its arithmetic is intrinsically more costly, and
- it causes substantially greater noise growth.

Each homomorphic multiplication consumes a level of the noise budget, and the accumulated noise along a path dictates the required parameter sizes. Consequently, the per-operation

latency in leveled FHE is closely tied to the *multiplicative depth* (MD) of the implementing circuit — the maximum number of sequential homomorphic multiplications along any PI to PO path. High-MD circuits either require very large parameters or incur bootstrapping, both of which inflate runtime. Minimizing MD has therefore become a central objective.

This emphasis on MD harmonizes naturally with Boolean modeling. In the binary domain, addition and multiplication correspond to XOR and AND, respectively; target computations can thus be represented as *XOR–AND–inverter graphs* (XAGs) and optimized using logic synthesis techniques aimed at depth reduction [41, 12, 112]. The appeal is further strengthened by the alignment between binary plaintext spaces and conventional software abstractions, which facilitates compilation pipelines that translate high-level programs into Boolean circuits for secure execution [42].

However, minimizing MD in isolation commonly increases the total number of multiplications, i.e., the circuit's *multiplicative complexity* (MC). Depth reduction techniques often trade a serial chain of multiplications for many parallel ones, inflating MC even as MD decreases. Since end-to-end circuit evaluation time in leveled FHE depends jointly on *both* MD (which drives parameter sizes and thus the cost of *all* operations) and MC (which counts the expensive multiplications actually executed), MD-only optimization can yield circuits that are theoretically shallow yet practically slower. Existing tools typically address the MD–MC tension in an ad hoc fashion — e.g., running MD-reduction passes until a heuristic MC threshold is exceeded — rather than optimizing the two criteria jointly. This reveals a methodological gap.

Contributions. We address this gap by formulating leveled-FHE circuit optimization as a *joint* problem over MD and MC, guided by an FHE cost proxy that captures

- parameter-growth penalties due to depth, and
- the intrinsic expense of multiplications.

Concretely, we introduce:

1. an exact logic synthesis engine for *small-scale* Boolean functions that explores the optimal MD–MC trade-off and returns FHE-cost-optimal realizations;
2. a scalable *MC-aware MD minimization* algorithm that invokes exact synthesis locally on critical cuts to optimize *large* circuits' FHE cost;
3. an integrated optimization framwork that composes our pass with the strongest MD-only reducer from the literature — *ESOP balancing* [95] — to systematically traverse the MD–MC design space.

While the precise form of the "FHE cost" remains an open modeling question, we adopt the empirically supported metric $\mathrm{MC} \times \mathrm{MD}^2$, derived from simplifying the analysis in [64],

as a practical instantiation; The framework remains agnostic and supports alternative user-specified models. Under this configuration, our experiments show substantial speedups in homomorphic evaluation over MD-optimized circuits, with an average improvement of 21.32% and a reduction of circuit optimization time by roughly four orders of magnitude, compared to the prior learning-and-rewriting approach proposed in [112].

The technical content of this chapter is based on the work published in [192].

3.2 Preliminaries

This section recalls the scheme-level distinctions relevant to this chapter (leveled vs. fast bootstrapping; Section 3.2.1), introduces the specific Boolean-network notations we use (Section 3.2.2), and summarizes prior work on accelerating homomorphic function evaluation via circuit optimization under leveled schemes (cf. Section 3.2.4). We conclude by motivating MC-aware MD minimization (see the "Significance of MC-Aware MD Minimization" sub-subsection in Section 3.2.4). General foundations (logic networks, logic synthesis, and the history of HE constructions) are covered in Chapter 2.

3.2.1 FHE Schemes: Leveled vs. Fast Bootstrapping

FHE schemes are often categorized by how they manage noise during homomorphic multiplications. Leveled schemes (e.g., BGV, BFV) provision a noise budget so that a circuit of *multiplicative depth* (MD) d can be evaluated without refreshing; supporting depth d typically requires $(d+1)$ modulus "layers" [134], as each multiplication consumes one layer. Once the budget is exhausted, further multiplications require bootstrapping, which is expensive and thus avoided whenever possible. The advantage is that, within the budget, all computation proceeds without bootstrapping; the drawback is that larger d forces larger parameters (keys/-ciphertexts and polynomial arithmetic), raising the cost of *every* operation. This trade-off makes circuit MD a primary driver of performance under leveled schemes and motivates the optimization focus of this chapter.

Fast-bootstrapping schemes (e.g., FHEW, TFHE) instead refresh noise frequently. Under these schemes, every gate within an FHE circuit is evaluated by a bootstrapping that both realize the logic of the gate and resets the noise. Since each refresh acts as a *lookup table* (LUT)-like evaluation on encrypted inputs, the bootstrap operation in this category of schemes is distinguished as *programmable bootstrap* (PBS) [57]. Consequently, runtime depends chiefly on the *number of bootstraps*, not on circuit MD, and complex Boolean functions can be collapsed into single PBS calls when representable by compact truth tables. Empirically, single PBS costs are on the order of a few to a few tens of milliseconds and largely independent of global MD [40]. These differences drive distinct optimization goals: leveled FHE prioritizes MD reduction (even at modest increases in gate count), whereas fast-bootstrapping schemes prioritize minimizing PBS invocations [90, 40, 196]. This chapter is devoted to the leveled

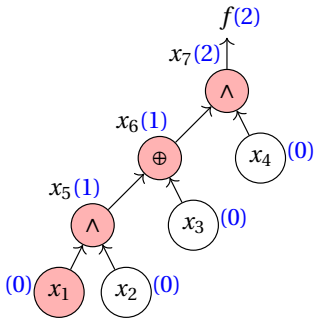


Figure 3.1: An XAG for the four-variable function with truth table #7800. Node MDs appear in blue; a critical PI→PO path is highlighted in red.

paradigm; fast-bootstrapping and its synthesis implications are treated in Part A, Chapters 4–5.

3.2.2 Multiplicative Depth in Logic Networks

We model a leveled FHE Boolean circuit as a logic network with node set V and edge set E: PIs I and logic gates G. In the binary plaintext setting, homomorphic addition and multiplication correspond to Boolean XOR and AND, respectively; thus, we use the standard XAG with two-input XOR/AND gates and optional input negation. We use *network* and *circuit* interchangeably.

Figure 3.1 depicts an XAG for a four-variable function (truth table #7800), with $\wedge$ and $\oplus$ denoting AND and XOR. Let $|I| < i \leq |I| + |G|$ index gate steps; each step is $x_i = x_{j_1} \circ_i x_{j_2}$ with $\circ_i \in \{\wedge, \oplus\}$ and $1 \leq j_1 < j_2 < i$. Define node-wise multiplicative depth (MD) recursively by

$$\sigma_i = \begin{cases} \max\{\sigma_{j_1}, \sigma_{j_2}\}, & \circ_i = \oplus, \\ \max\{\sigma_{j_1}, \sigma_{j_2}\} + 1, & \circ_i = \wedge. \end{cases} \tag{3.1}$$

PIs have $\sigma = 0$. The circuit MD is $d = \max\{\sigma_i\}$. Any PI→PO path achieving d is a *critical path*. The circuit's *multiplicative complexity* (MC) [34] is the number of AND gates

$$c = |\{i : \circ_i = \wedge\}|. \tag{3.2}$$

3.2.3 Circuit Optimization in Leveled FHE Schemes

To evaluate a Boolean function on encrypted data, the target function is compiled to a circuit compatible with the chosen scheme. Under leveled FHE schemes, three lines of work are particularly relevant.

3.2.4 MD Reduction

Early efforts reused depth-optimization in ABC [37] to indirectly reduce MD [42]. Carpov *et al.* [41] then introduced algebraic, structure-aware rewrites (e.g., re-association on critical paths) tailored to MD. The tool LOBSTER [112] combines offline rule learning from training circuits and online maximal rewriting to minimize MD; despite strong depth results, its learning and matching stages can be costly.

A technique less publicized in crypto but highly effective for MD is *ESOP balancing* [95], originally for T-depth minimization in fault-tolerant quantum computing. Each ESOP cube is realized as a balanced AND tree; cubes are combined by XORs (which do not increase MD), yielding low-MD XAGs. ESOP balancing exploits Boolean-specific structure and is therefore applicable only to Boolean circuits. We will compose with ESOP balancing in our integrated flow (Section 3.5.2).

Arithmetic Circuit vs. Boolean Circuit

An orthogonal strand focuses on word-level/datapath optimizations. Many applications leverage arithmetic over large plaintext moduli in BGV/BFV or approximate real arithmetic in CKKS [52], performing more work per homomorphic operation (e.g., one ciphertext addition vs. 32 bit-level adders). Appropriate algorithm choices and arithmetic circuit designs can yield substantial speedups [88, 179]. These high-level optimizations are complementary to the Boolean gate-level focus of this chapter. In practice, an FHE compiler can apply word-level transformations first, then invoke the Boolean backend presented here to optimize the remaining bit-level components.

Automatic Bootstrapping Management

For workloads with large MD (e.g., private neural network inference), selectively bootstrapping to refresh noise may be beneficial [53]. Deciding where to bootstrap is the *automatic bootstrapping management* problem. Due to complexity, most work treats it separately from MD optimization, typically assuming a pre-set MD bound [114, 142]. In this chapter, we confine ourselves to leveled circuit design and do not manage bootstrapping.

Because the underlying schemes differ, the "bootstrapping management" addressed in Part A, Chapters 4–5 is semantically distinct from the leveled setting considered here: in leveled schemes, bootstrapping serves solely as a noise-refresh operation, whereas in fast-bootstrapping schemes it additionally *evaluates a small function* and — crucially for Chapter 5 — can *change the ciphertext space*. Consequently, the scheduling problems in those chapters optimize the placement and number of bootstrapping invocations and space switches, rather than enforcing a depth budget as in leveled FHE.

Significance of MC-Aware MD Minimization

Lower MD at the expense of increased MC raises the total count of homomorphic multiplications, potentially negating depth gains. Effective leveled-FHE optimization must therefore account for *both* MD and MC. This poses two challenges:

(1) designing synthesis and optimization techniques that *jointly* optimize MD and MC, and

(2) formulating a cost model — the *FHE cost* [1] — that captures how depth affects parameter size and how multiplication counts affect runtime.

The second remains open and may vary by scheme and parameter set [64]; e.g., for BFV, ciphertext size has been fit by a power law in depth d: $l = 1.2215 \cdot d^{2.0179}$ [64], while the asymptotic bit complexity of homomorphic multiplication aligns with arbitrary-precision integer multiplication $\mathcal{O}(n \log n \log\log n)$ [12].

For concreteness in evaluation, we adopt $\mathrm{MC} \times \mathrm{MD}^2$ as a working FHE cost. Crucially, our framework is *cost-agnostic*: users may provide any $\kappa(\mathrm{MC}(G), \mathrm{MD}(G))$ for an XAG G. This flexibility supports both deployment-specific tuning and future research toward sharper cost models.

The remainder of the chapter develops exact and scalable methods for joint MD-MC optimization and integrates them with state-of-the-art MD reduction (ESOP balancing), while Part A, Chapters 4–5 adapt the cone-identification viewpoint to the TFHE setting.

3.3 FHE-Cost-Optimum Synthesis for Boolean Functions

This section addresses the exact-synthesis problem:

Given a Boolean function with a small number of inputs and a specified FHE cost metric that depends on both MC and MD, how can we synthesize an exactly *FHE-cost-optimal circuit (XAG) implementation?*

We formulate an exact method that is parameterized by target MC and MD and leverages two abstractions — the *AND fence* and the *abstract XAG* — so that cost is driven solely by (MC, MD) while functional feasibility is decided by *Boolean satisfiability* (SAT).

[1] In this chapter, "FHE cost" always denotes a leveled-FHE cost model; we drop "leveled" for brevity. TFHE-specific costs are treated in Chapters 4–5.

3.3.1 Overview of the Methodology

AND Fence

An XAG's *AND fence* compactly summarizes how AND gates populate MD levels [195]. Given an XAG of MD d, we define the fence as

$$\mathscr{F} = (c_1, c_2, \ldots, c_d),$$

where $c_i \in \mathbb{Z}_{\geq 0}$ counts AND gates whose node-wise MD equals i. As a consequence,

$$\mathrm{MC}(\mathscr{F}) = \sum_{i=1}^{d} c_i \qquad \text{and} \qquad \mathrm{MD}(\mathscr{F}) = d. \tag{3.3}$$

For example, the fence of Figure 3.1 is $(1,1)$. In what follows, we attribute MC and MD directly to the fence $\mathscr{F}$. Because the FHE cost used in this chapter depends only on $(\mathrm{MC}, \mathrm{MD})$, $\mathscr{F}$ suffices to determine cost — independently of the XOR interconnections.

Abstract XAG

An *abstract XAG* generalizes an XAG by replacing chains of two-input XORs with flexible *XOR clouds* (XOR nodes of arbitrary nonzero fan-in) [163]. A single-input XOR acts as a buffer.

In the original form, an abstract XAG satisfies:

(i) Each AND has two XOR-cloud fan-ins;

(ii) Each XOR-cloud fan-in is either a PI or the output of a lower-*logic*-level AND;

(iii) Each PO is an XOR cloud.

We enhance MD-awareness with two constraints that make the per-step MD explicit:

(i) Every fan-in to step x_i comes from a strictly lower *MD* level, i.e., $\forall x_j \in L_{i,1} \cup L_{i,2} : \sigma_j < \sigma_i$;

(ii) At least one fan-in originates from the immediately preceding MD level, i.e., $\exists x_j \in L_{i,1} \cup L_{i,2} : \sigma_j = \sigma_i - 1$.

With these constraints, the *topology* of an abstract XAG is determined by the fence $\mathscr{F} = (c_1, \ldots, c_d)$: there are c_1 steps at MD 1, c_2 at MD 2, and so on, with (i) and (ii) constraining permissible fan-ins per step.

Each abstract-XAG step consists of a two-input AND fed by two XOR clouds. Hence, the number of steps equals the MC c. For an n-input function f, step x_i $(n < i \leq n + c)$ is

$$x_i = \Big(\bigoplus_{x_j \in L_{i,1}} x_j \Big) \wedge \Big(\bigoplus_{x_j \in L_{i,2}} x_j \Big), \qquad 1 \leq j < i, \tag{3.4}$$

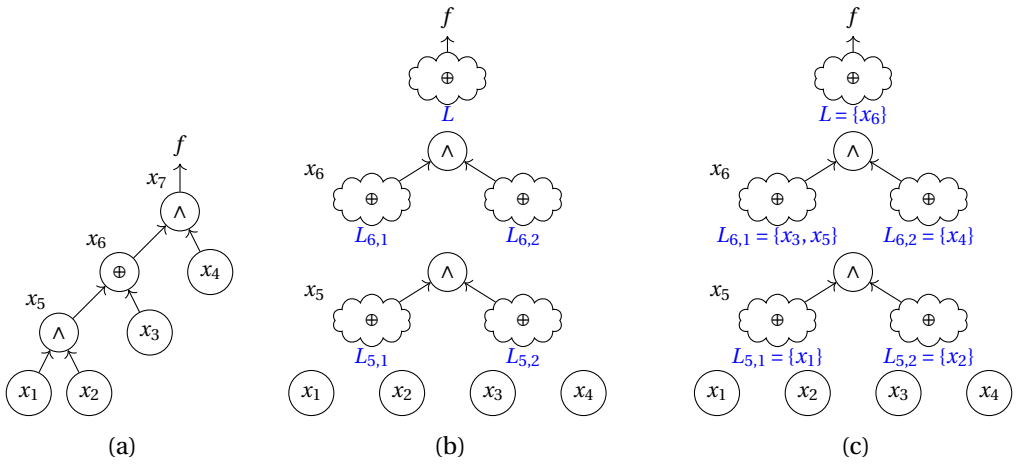


Figure 3.2: Deriving an abstract XAG from an XAG in two steps. (a) The original XAG (reproducing Figure 3.1); (b) topology induced by fence $\mathscr{F} = (1,1)$; (c) fan-in configuration of XOR clouds matching the original function (explicit lists shown in blue).

and the PO XOR cloud realizes f as a linear form

$$f = \bigoplus_{x_i \in L} x_i, \qquad 1 \le i \le n + c. \tag{3.5}$$

Example 3.3.1. *Given the XAG in Figure 3.2a with fence $\mathscr{F} = (1,1)$, we deduce $\sigma_5 = 1$ and $\sigma_6 = 2$, which determines the abstract-XAG topology in Figure 3.2b. By configuring XOR-cloud fan-ins accordingly, we obtain a functionally equivalent abstract XAG in Figure 3.2c.*

The XAG→abstract-XAG conversion above is deterministic; the inverse is not unique because several XOR-cloud configurations can implement the same function with the *same* fence. A canonical XAG can be recovered by decomposing each XOR cloud into two-input XORs; since (MC, MD) is preserved, any FHE-cost-optimum abstract XAG yields an FHE-cost-optimum XAG after decomposition. Thus, the search for a cost-optimum XAG reduces to testing abstract-XAG feasibility for a fence.

Sketch of Methodology

We rely on three observations:

(1) An XAG's FHE cost depends only on its fence $\mathscr{F}$ via $(\mathrm{MC}(\mathscr{F}), \mathrm{MD}(\mathscr{F}))$.

(2) For a given function f and fence $\mathscr{F}$, feasibility of an abstract XAG implementing f can be decided by SAT.

(3) Decomposing an abstract XAG to a concrete XAG preserves (MC, MD) and hence FHE cost.

We therefore enumerate fences in ascending FHE cost and, for each $\mathscr{F}$, solve a SAT instance. The first satisfiable instance yields the FHE-cost-optimum abstract XAG (and thus XAG) for f.

3.3.2 SAT Encoding

We encode the feasibility question — "does there exist an abstract XAG with fence $\mathscr{F}$ that realizes f?" — into SAT.

Variables

We use two classes of Boolean variables:

Selection variables.

(i) s_{ijk} for $n < i \le n+c$, $1 \le j < i$, and $k \in \{1,2\}$, where $s_{ijk} = \mathsf{true}$ iff x_j is an input to the k-th XOR cloud of step x_i (i.e., $x_j \in L_{i,k}$).

(ii) s_j for $1 \le j \le n+c$, where $s_j = \mathsf{true}$ iff x_j is an input to the PO XOR cloud (i.e., $x_j \in L$).

Function variables. f_{jl} for $1 \le j \le n+c$ and $0 \le l \le 2^n - 1$. Let $(b_n, \ldots, b_1)_2$ be the binary expansion of l. For $j \le n$, $f_{jl} = b_j$ encodes PI values. For $j > n$, $f_{jl} = \mathsf{true}$ indicates that step x_j evaluates to true on the PI assignment $(x_1, \ldots, x_n) = (b_1, \ldots, b_n)$.

Clauses

We enforce correctness, non-emptiness of XOR clouds, and MD-aware fan-in constraints.

Clause 1 (step semantics). For each $n < i \le n+c$ and each l,

$$f_{il} \leftrightarrow \Big(\bigoplus_{j=1}^{i-1}(s_{ij1} \wedge f_{jl})\Big) \wedge \Big(\bigoplus_{j=1}^{i-1}(s_{ij2} \wedge f_{jl})\Big).$$

Clause 2 (PO function). For each l, let $f(l) \in \{0,1\}$ be the truth-table value of f at index l. We require

$$f(l) = \bigoplus_{j=1}^{n+c}(s_j \wedge f_{jl}).$$

(The left-hand side is a constant bit; thus we use '=' rather than '$\leftrightarrow$'.)

Clause 3 (non-empty XOR clouds).

$$\bigvee_{j=1}^{i-1} s_{ij1} \quad \wedge \quad \bigvee_{j=1}^{i-1} s_{ij2} \qquad (n < i \le n+c), \qquad \text{and} \qquad \bigvee_{j=1}^{n+c} s_j.$$

Clause 4 (fan-ins from lower MD levels). For each step x_i of MD σ_i,

$$s_{ijk} = \mathsf{false} \quad \text{whenever} \quad \sigma_j \ge \sigma_i, \qquad (n < i \le n+c,\ k \in \{1,2\}).$$

This enforces constraint (i).

Clause 5 (at least one fan-in from the immediately preceding level). For each step x_i of MD σ_i,

$$\bigvee_{\substack{1 \le j < i \\ \sigma_j = \sigma_i - 1}} \left(s_{ij1} \vee s_{ij2}\right).$$

This enforces constraint (ii).

The resulting formula is converted to CNF via Tseitin encoding [173] and solved. A satisfiable instance certifies the existence of an abstract XAG (with fence $\mathscr{F}$) implementing f.

3.3.3 Identification of AND Fence Candidates

To guarantee optimality, we must consider all fences that could yield a lower FHE cost than a known reference. Enumerating every Boolean function is infeasible: for $n = 5$ there are 2^{32} functions. We therefore apply *affine classification* [77] to work with class representatives. Since affine-equivalent functions share the same MC minimum [172] and, when inputs have equal MD, preserve MD as well, we can use representatives to bound the search space.

Theorem 3.3.1 (MD preservation under affine equivalence)**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be implemented by an XAG, and suppose all primary inputs have the same MD σ_0. If g is* affine-equivalent *to f, i.e., there exist an invertible linear map $A \in \mathbb{B}^{n\times n}$ and vectors $u \in \{0,1\}^n$, $v \in \{0,1\}$ such that*

$$g(x) = f(Ax \oplus u) \oplus v,$$

then, the minimum achievable MD of g equals that of f.

Proof. We first note two facts about XAGs:

(i) XOR gates do not increase MD (they take the maximum of their fan-in MDs), and

(ii) AND gates increase MD by exactly one.

We denote by $\mathrm{MD}(h)$ the minimum MD over all XAGs that implement a Boolean function h under the given input MD assignment. Throughout, all n inputs are assumed to enter the circuit at the same MD σ_0.

(1) $\mathrm{MD}(g) \le \mathrm{MD}(f)$**.** Start from an MD-optimal XAG for f. Prepend an *input linear layer* that computes $y = Ax \oplus u$ using only XORs (and constants), producing the n signals $y_1, \ldots, y_n$. Since all inputs have MD σ_0 and XOR does not increase MD, each y_i also has MD σ_0. Feed y into the original (nonlinear) part of the f-circuit; this realizes $f(Ax \oplus u)$. Finally, append a single XOR with the constant v to obtain $g(x)$. The added linear layers do not increase MD anywhere, hence the overall MD is the same as that of the original f-circuit. Therefore $\mathrm{MD}(g) \le \mathrm{MD}(f)$.

(2) $\mathrm{MD}(f) \le \mathrm{MD}(g)$**.** Because A is invertible over $\mathbb{B}$, $x = A^{-1}(y \oplus u)$ and $f(x) = g(A^{-1}(x \oplus u)) \oplus v$. Apply the same construction as above but with A^{-1} and u: prepend the XOR-only layer $y = A^{-1}(x \oplus u)$, evaluate g on y, then XOR with v. Again, the linear layers do not increase MD, so any implementation of g with MD d yields an implementation of f with MD at most d. Hence $\mathrm{MD}(f) \le \mathrm{MD}(g)$.

Combining (1) and (2) gives $\mathrm{MD}(f) = \mathrm{MD}(g)$, proving MD preservation under affine equivalence when all inputs share the same initial MD σ_0.

Relation to generators. The standard generators of affine transformations — input permutations, input/output negations, translations: $x_i \leftarrow x_i \oplus x_j$, and the disjoint translation that XORs an input directly into the output — are all XOR-only modifications. With equal input MDs, these generators preserve the per-signal MD at the input to every AND gate, so the maximal count of AND levels on any PI→PO path (the circuit MD) is unchanged. □

For $n = 5$, there are 48 affine classes [174], and the MC-minimum XAG per representative is known [163]. For any 5-variable function f, let (c_r, d_r) denote the MC and MD of its MC-minimum implementation (obtained from its representative). Any strictly better FHE-cost implementation must satisfy

$$c \ge c_r, \qquad d < d_r, \qquad c \cdot d^2 < c_r \cdot d_r^2.$$

Given target (c, d), the set of compatible fences is

$$\mathscr{F}(c,d) = \Big\{(c_1, \ldots, c_d) \in \mathbb{Z}_{\ge 0}^d \;\Big|\; \sum_{i=1}^{d} c_i = c,\;\; c_d \ge 1\Big\},$$

i.e., the weak compositions of c into d parts with at least one AND at level d. Enumerating $\mathscr{F}(c,d)$ reduces to a standard integer-partition/composition problem [106]. Considering all $\mathscr{F} \in \mathscr{F}(c,d)$ for the admissible (c,d) pairs ensures that, if a lower-cost implementation exists, it will be discovered by our SAT feasibility check.

Algorithm 3.1: Exact synthesis of an FHE-cost-optimum XAG for a Boolean function

Input: Boolean function f; MC c_r and MD d_r of an MC-minimum XAG for f.
Output: FHE-cost-optimum XAG $N^\star$ for f.

$\kappa \leftarrow c_r \cdot d_r^2$ // current best FHE cost
$N^\star \leftarrow$ XAG of cost κ (the MC-minimum design)
$d \leftarrow d_r$
while $d > 1$ **do**
  $d \leftarrow d - 1$ // attempt to reduce depth
  $c_{\max} \leftarrow \lfloor \kappa / d^2 \rfloor$ // cost admissibility bound
  improved $\leftarrow$ false
  for $c \leftarrow c_r$ **to** $c_{\max}$ **do**
    foreach $\mathscr{F} \in \mathscr{F}(c, d)$ **do**
      $\zeta \leftarrow$ SATENCODING$(f, \mathscr{F})$
      $N' \leftarrow$ SATSOLVE(ζ) // returns abstract XAG or NULL
      if $N' \neq$ NULL **then**
        $\kappa \leftarrow c \cdot d^2$
        $N^\star \leftarrow$ DECOMPOSEXORCLOUDS(N')
        improved $\leftarrow$ true
        break // move to smaller MD
    if *improved* **then**
      break
  if *not* *improved* **then**
    return $N^\star$ // no feasible fence at MD d; prior $N^\star$ is optimum
return $N^\star$

3.3.4 Exact Synthesis Paradigm for Boolean Functions

Algorithm 3.1 synthesizes an FHE-cost-optimum XAG. Starting from the MC-minimum realization (cost $\kappa = c_r d_r^2$), it attempts to reduce depth to $d_r - 1$, enumerating admissible MC values and fences in ascending cost. Upon finding a feasible abstract XAG at depth d, it updates the incumbent solution and proceeds to depth $d - 1$. If no fence at depth d is feasible, the previously found solution $N^\star$ is optimal.

A natural alternative is to enumerate fences directly by increasing FHE cost. Empirically, however, MD-first scheduling avoids a surge of unsatisfiable SAT instances that often occurs when "jumping" across (c, d) pairs in pure cost order, and Theorem 3.3.2 guarantees no loss of optimality:

Theorem 3.3.2 (Monotonicity in MD)**.** *If no abstract XAG exists for a Boolean function f with any fence $\mathscr{F}_1$ of MD d, then no abstract XAG exists for f with any fence $\mathscr{F}_2$ of MD $d - 1$.*

Proof. We prove the contrapositive. Assume there exists an abstract XAG N_2 that implements f with fence $\mathscr{F}_2 = (c_1, \ldots, c_{d-1})$ of MD $d - 1$. We construct an abstract XAG N_1 of MD d that still implements f.

Let r denote the (unique) PO signal of N_2. Form N_1 by *adding one final step* at local MD level d whose two XOR-cloud fan-ins are both the singleton $\{r\}$, i.e., the new step computes

$$r' = r \wedge r = r.$$

This preserves the overall Boolean function (since $r \wedge r \equiv r$), increases the depth by exactly one, and respects the abstract-XAG constraints: each fan-in comes from a strictly lower MD level and at least one comes from the immediately preceding level. The resulting fence is $\mathscr{F}_1 = (c_1, \ldots, c_{d-1}, 1)$, which has MD d.

Thus, if a realization exists at depth $d-1$, a realization also exists at depth d. Taking the contrapositive, if no abstract XAG exists at MD d, then none exists at MD $d-1$. □

3.4 Exact Synthesis for Sub-circuits

While the exact synthesis method of Section 3.3 yields *optimum* implementations for small Boolean functions, scalability limits its direct use on larger instances. To produce high-quality designs for practical networks, we lift the granularity from whole functions to *cuts* (sub-circuits): we replace sub-circuits along the critical path by locally optimal implementations, thereby improving the global circuit.

3.4.1 Effects of Non-zero Input MD

Our global goal is to minimize the (leveled) FHE cost. Locally, we optimize cuts rooted on the current critical path. For a cut, we prioritize minimizing the MD of its *root*, while constraining MC via the admissible fences collected in Section 3.3.3. This is consistent with MC awareness: we exclude candidates that would drastically inflate local MC; we also avoid relying on potentially brittle global sharing to "pay back" local MC increases.

Crucially, treating a cut's *local function* as a stand-alone Boolean function and applying Section 3.3 does *not* generally minimize the root MD of the cut. The reason is that the cut's leaves are usually *internal* nodes with *non-zero* MDs, not PIs. Thus, the present problem generalizes Section 3.3: there, all leaves have MD 0.

For an n-leaf cut $\mathscr{C}$, we write its *input MD vector* as

$$\mathscr{L} = (\sigma_1, \ldots, \sigma_n),$$

where σ_i is the MD of the i-th leaf. We call $\mathscr{L}$ *balanced* if all entries are equal, and *imbalanced* otherwise. The optimal implementation of a cut *depends* on $\mathscr{L}$ even when its local function is fixed, as the example in Figure 3.3 shows.

To analyze this dependence rigorously, we work in the abstract XAG model (cf. Section 3.3.1),

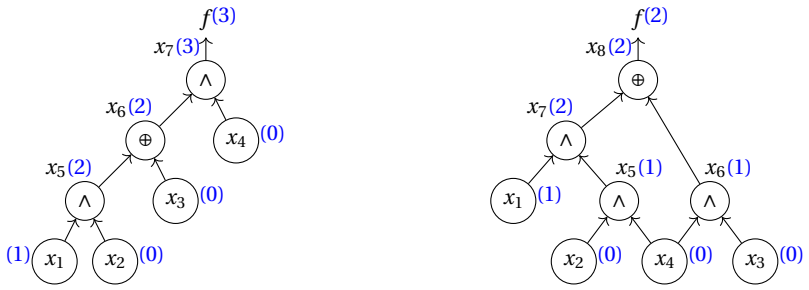


Figure 3.3: Two XAGs, N_1 and N_2, implementing the same four-leaf cut $\mathscr{C}$ with local function #7800 under input MD $\mathscr{L} = (1,0,0,0)$. Node MDs are annotated in blue. (a) N_1 has fence $\mathscr{F}_1 = (1,1)$, MC = 2, MD = 2; (b) N_2 has fence $\mathscr{F}_2 = (2,1)$, MC = 3, MD = 2. Although N_1 is cheaper as a *stand-alone* implementation of the Boolean function (lower MC), the root MD under $\mathscr{L}$ is 3 for N_1 and 2 for N_2, making N_2 locally preferable for this cut.

which preserves (MC, MD). Let $\mathscr{C}$ have leaves $x_1, \ldots, x_n$ with input MDs $\mathscr{L} = (\sigma_1, \ldots, \sigma_n)$. Consider a c-step abstract XAG with fence $\mathscr{F} = (c_1, \ldots, c_d)$ (so $c = \sum_\ell c_\ell$ and $\mathrm{MD}(\mathscr{F}) = d$). Define the *local MD* $\delta(v)$ of a node v as

$$\delta(v) = d - (\#\,\text{AND levels from } v \text{ to the root}),$$

so the root has $\delta_{\text{root}} = d$. For each leaf x_i, let

$$\Delta_i = \delta_{\text{root}} - \delta(x_i)$$

be the maximum number of AND levels on any path from x_i to the root (equivalently, the largest local-depth "distance" used by x_i). Then the MD of the root is

$$\sigma_{\text{root}} = \max_{1 \le i \le n} \{\sigma_i + \Delta_i\}, \tag{3.6}$$

with bounds $0 \le \Delta_i \le d$. The lower bound corresponds to using x_i only at the PO XOR; the upper bound is realized if x_i first appears at local level 1.

Balanced inputs. If $\mathscr{L}$ is balanced, say $\sigma_i = a$ for all i, then

$$\sigma_{\text{root}} = a + d = a + \mathrm{MD}(\mathscr{F}), \tag{3.7}$$

independent of which leaf is critical. Thus, for balanced $\mathscr{L}$, minimizing the cut's root MD under fence $\mathscr{F}$ reduces to minimizing d; no per-leaf scheduling is required.

Imbalanced inputs. When $\mathscr{L}$ is imbalanced, the identity of the critical leaf depends on how the leaves are *scheduled* into local MD levels; see Eq. (3.6). Hence, we must extend the SAT

encoding of Section 3.3.2 to encode per-leaf availability across local MD levels.

3.4.2 Integrating Scheduling into SAT Encoding

For a fixed fence $\mathscr{F}$ of depth d, we associate a *scheduling vector*

$$\mathscr{S} = (\Delta_1, \ldots, \Delta_n) \in \{0, 1, \ldots, d\}^n,$$

where Δ_i is as in Eq. (3.6). Equivalently, define the *availability level*

$$\ell_i = d - \Delta_i \in \{0, 1, \ldots, d\},$$

the smallest local MD at which leaf x_i is permitted to feed a step (with $\ell_i = 0$ meaning "PO only"). We refine the SAT encoding by forbidding illegal fan-ins: *Scheduling clause (per leaf).* Let $\mathscr{V}_\ell$ be the set of abstract-XAG steps at local MD $\ell \in \{1, \ldots, d\}$ (determined by $\mathscr{F}$). For each leaf x_j with availability ℓ_j,

$$\forall \ell < \ell_j,\ \forall i \in \mathscr{V}_\ell,\ \forall k \in \{1, 2\}: \quad s_{ijk} = \mathsf{false}.$$

This ensures that x_j cannot feed steps below its availability level.

Fixing $(\mathscr{F}, \mathscr{S})$ determines the target root MD

$$\sigma_{\text{root}}(\mathscr{F}, \mathscr{S}, \mathscr{L}) = \max_i \{\sigma_i + \Delta_i\}.$$

We then iterate SAT instances in increasing σ_{root} until the first satisfiable instance; its abstract-XAG witness is decomposed to obtain the XAG.

3.4.3 Strategic Selection of Scheduling Solutions

Enumerating all schedules is prohibitive: there are $(d+1)^n$ possibilities for a fence of depth d. We therefore

(i) choose a *promising initial schedule,* then

(ii) relax it minimally if infeasible, increasing σ_{root} by one each time.

Throughout, assume $\mathscr{L}$ is sorted as $\sigma_1 \leq \cdots \leq \sigma_n$.

Choosing the Initial Schedule

Intuitively, we would like each leaf to appear as late as possible (large Δ_i), thereby minimizing σ_{root}. However, each local MD level must have *enough fan-in candidates* (leaves or outputs

Table 3.1: Nonlinear functions realizable by one abstract-XAG step as a function of available fan-in candidates.

# candidates	2	3	4	5
# nonlinear functions	1	9	55	285

from lower levels) to realize the required number of *nonlinear* functions at that level; otherwise the SAT instance is trivially infeasible.

Example 3.4.1. *Consider a* 4*-leaf cut with* $\mathscr{L} = (0,0,0,1)$ *and fence* $\mathscr{F}_1 = (2)$ *(two steps at local MD* 1*). If we schedule only* x_1, x_2 *to be available at level* 1 *(i.e.,* $\mathscr{S}_1 = (1,1,0,0)$*), then either both steps compute* $x_1 \wedge x_2$ *or one step degenerates to a linear function over* $\{x_1, x_2\}$ *— effectively reducing to fence* $\mathscr{F}_2 = (1)$*, which would have been explored earlier. This instance is therefore infeasible without solving SAT. Scheduling* $\mathscr{S}_2 = (1,1,1,0)$ *admits distinct nonlinear choices for both steps and avoids the trivial reduction.*

We estimate the minimum number of *distinct* nonlinear functions realizable at a level as a function of the number of candidates, by counting choices for the two XOR-cloud inputs $L_{i,1}, L_{i,2}$ (excluding linear cases $L_{i,1} = L_{i,2}$; quotienting by commutativity; and eliminating subset-redundant pairs [39]). For up to five candidates, the results are reported in Table 3.1.

Armed with this bound, we choose the initial schedule as follows: for each local level $\ell = 1, \ldots, d$, ensure that the cumulative pool (scheduled leaves for level ℓ plus outputs from levels $< \ell$) is large enough to support the c_ℓ steps; then schedule any remaining leaves *as late as possible* (namely, as soon as possible relative to the current σ_{root}) without creating a unique critical path. This process of selecting the initial schedule for a given fence is summarized as Algorithm 3.2.

Relaxing the Schedule (Next Schedule Candidate)

If the SAT instance for $(\mathscr{F}, \mathscr{S})$ is unsatisfiable, we relax the schedule by allowing leaves to appear *one level earlier* (thereby increasing σ_{root} by one). Concretely, if $\mathscr{S}_1$ yields target σ_{root}, the next schedule is

$$\mathscr{S}_2 = (\min\{d,\ \sigma_{\text{root}} + 1 - \sigma_1\}, \ldots, \min\{d,\ \sigma_{\text{root}} + 1 - \sigma_n\}),$$

unless all entries are already d (in which case no $(\mathscr{F}, \mathscr{S})$ implementation exists). This mirrors the second phase (lines 13–14) of Algorithm 3.2.

3.4.4 Exact Synthesis Paradigm for Sub-circuits

The procedure first computes the stand-alone optimum for $\mathscr{C}.f$ (Section 3.3); if $\mathscr{L}$ is balanced, Eq. (3.7) ensures optimality and we return. Otherwise, we use its root MD under $\mathscr{L}$ as an upper bound. For each fence, we derive an initial schedule and the associated target root MD;

Algorithm 3.2: Selecting an initial schedule for fence $\mathscr{F}$

Input: Fence $\mathscr{F} = (c_1, \ldots, c_d)$; input MD $\mathscr{L} = (\sigma_1, \ldots, \sigma_n)$ (sorted).
Output: Schedule $\mathscr{S} = (\Delta_1, \ldots, \Delta_n)$; target root MD σ_{root}.

$S \leftarrow \{1, \ldots, n\}$ `// indices of unscheduled leaves`
#cand $\leftarrow 0$ `// current candidate pool size`
for $\ell \leftarrow 1$ **to** d **do**
  $m \leftarrow \text{REQUIREDCANDIDATES}\left(\sum_{t=1}^{\ell} c_t\right)$ `// from Table 3.1`
  #need $\leftarrow \max\{0,\ m - \#\text{cand}\}$
  if #need > 0 **then**
    `// schedule the first #need leaves to be available at level` ℓ
    choose $S' \subseteq S$ with $|S'| =$ #need (smallest indices)
    foreach $j \in S'$ **do**
      $\Delta_j \leftarrow d - \ell$
      $S \leftarrow S \setminus \{j\}$
      #cand $\leftarrow$ #cand $+ 1$
  #cand $\leftarrow$ #cand $+ c_\ell$ `// outputs from level` ℓ
`// compute current target root MD from the scheduled prefix`
$J \leftarrow \{1, \ldots, n\} \setminus S$
$\sigma_{\text{root}} \leftarrow \max_{j \in J}\{\sigma_j + \Delta_j\}$
`// ASAP for remaining leaves without lowering availability below PO-only`
foreach $i \in S$ **do**
  $\Delta_i \leftarrow \min\{d,\ \sigma_{\text{root}} - \sigma_i\}$
return $(\mathscr{S}, \sigma_{\text{root}})$

we then iterate over increasing σ_{target}, solving the extended SAT instance for each $(\mathscr{F}, \mathscr{S})$ at that target. The first satisfiable instance yields the locally optimal implementation; if none is found below the upper bound, we return the function-optimal design.

Correctness note. Fix a fence $\mathscr{F}$ and input MD $\mathscr{L}$. The extended encoding ensures that a SAT witness for schedule $\mathscr{S}$ achieves $\sigma_{\text{root}}(\mathscr{F}, \mathscr{S}, \mathscr{L})$. Enumerating schedules in increasing σ_{root} therefore returns a solution with *minimum* root MD if one exists for that fence; minimizing over fences via the outer loop preserves optimality.

3.4.5 Classifying Exact-Synthesis Queries

Optimizing an entire network triggers many cut-synthesis queries. Many are equivalent and can reuse results. As in Section 3.3, we use Boolean classification, but under imbalanced $\mathscr{L}$, affine equivalence need not preserve MD. We therefore adopt *NPN classification* [87] and augment it with an *input-MD signature* so that two queries are identical iff

(i) their local functions share an NPN representative, and

Algorithm 3.3: Exact synthesis of an MD-minimal implementation for a cut

Input: Cut $\mathscr{C}$ with local function $\mathscr{C}.f$ and input MD $\mathscr{C}.\mathscr{L}$; library of admissible fences *lib*.
Output: Locally optimal XAG N for $\mathscr{C}$.

$N \leftarrow$ ExactSynthesisForFunction($\mathscr{C}.f$) // Sec. 3.3
if IsBalanced($\mathscr{C}.\mathscr{L}$) **then**
  return N // Eq. (3.7)
$\sigma_{\text{root}} \leftarrow$ ComputeRootMD($N, \mathscr{C}.\mathscr{L}$) // upper bound
foreach $\mathscr{F} \in lib$ **do**
  $(\mathscr{F}.\mathscr{S}, \mathscr{F}.\sigma_{\text{root}}) \leftarrow$ InitialSchedule($\mathscr{F}, \mathscr{C}.\mathscr{L}$)
$\sigma_{\text{target}} \leftarrow \min_{\mathscr{F} \in lib} \mathscr{F}.\sigma_{\text{root}}$
while $\sigma_{target} < \sigma_{root}$ **do**
  foreach $\mathscr{F} \in lib$ **do**
    if $\mathscr{F}.\sigma_{root} = \sigma_{target}$ **then**
      $\zeta \leftarrow$ ExtendedSatEncoding($\mathscr{C}.f, \mathscr{F}, \mathscr{F}.\mathscr{S}$)
      $N' \leftarrow$ SatSolve(ζ)
      if $N' \neq$ NULL **then**
        return DecomposeXorClouds(N')
      $(\mathscr{F}.\mathscr{S}, \mathscr{F}.\sigma_{\text{root}}) \leftarrow$ UpdateSchedule($\mathscr{F}, \mathscr{F}.\mathscr{S}$)
  $\sigma_{\text{target}} \leftarrow \sigma_{\text{target}} + 1$
return N // fall back to function-optimal design

(ii) their input-MD signatures match.

We generate signatures in two steps:

Lemma 3.4.1 (Alignment)**.** *Let $\mathscr{C}_1, \mathscr{C}_2$ be n-cuts with the same local function and input MDs $\mathscr{L}_1 = (\sigma_1, \ldots, \sigma_n)$ and $\mathscr{L}_2 = (\sigma'_1, \ldots, \sigma'_n)$. If $\sigma_i - \sigma'_i$ is a constant a for all i, then the two cuts have identical optimal implementations.*

Proof. Fix any fence $\mathscr{F}$ of depth d and any schedule $\mathscr{S} = (\Delta_1, \ldots, \Delta_n) \in \{0, \ldots, d\}^n$ (the earliest local-MD level at which each leaf may be used). For an input-MD vector $\mathscr{L}$, the root MD realized by $(\mathscr{F}, \mathscr{S})$ equals

$$\sigma_{\text{root}}(\mathscr{F}, \mathscr{S}; \mathscr{L}) = \max_{1 \le i \le n} \{\sigma_i + \Delta_i\}.$$

If $\sigma'_i = \sigma_i - a$ for all i, then

$$\sigma_{\text{root}}(\mathscr{F}, \mathscr{S}; \mathscr{L}_2) = \max_i \{\sigma'_i + \Delta_i\} = \max_i \{\sigma_i + \Delta_i - a\} = \sigma_{\text{root}}(\mathscr{F}, \mathscr{S}; \mathscr{L}_1) - a.$$

Hence, for any two candidate implementations N_1 and N_2 (each induced by some $(\mathscr{F}_1, \mathscr{S}_1)$ and $(\mathscr{F}_2, \mathscr{S}_2)$), the ordering of their root MDs is preserved under the shift from $\mathscr{L}_1$ to $\mathscr{L}_2$:

$$\sigma_{\text{root}}(N_1; \mathscr{L}_1) \le \sigma_{\text{root}}(N_2; \mathscr{L}_1) \iff \sigma_{\text{root}}(N_1; \mathscr{L}_2) \le \sigma_{\text{root}}(N_2; \mathscr{L}_2).$$

Therefore, the set of minimizers (optimal implementations) is the same for $\mathscr{L}_1$ and $\mathscr{L}_2$. □

Alignment subtracts the minimum element from $\mathscr{L}$.

Lemma 3.4.2 (Dominance)**.** *Let $\mathscr{C}_1$ be an n-cut with input MD $\mathscr{L}_1 = (\sigma_1, \ldots, \sigma_n)$ (sorted) and optimal XAG MD d. If there exists i with $\sigma_i - \sigma_{i-1} \geq d$, then the optimal implementation for $\mathscr{C}_1$ is also optimal for any n-cut $\mathscr{C}_2$ with the same local function and input MD $\mathscr{L}_2 = (0, \ldots, 0, \sigma_{i+1} - \sigma_i, \ldots, \sigma_n - \sigma_i)$.*

Proof. Let d be the minimum root MD achievable for $\mathscr{C}_1$. Consider any candidate implementation (fence and schedule), and let Δ_a be the maximum number of AND levels from leaf x_a to the root (so $0 \leq \Delta_a \leq d$). The root MD under $\mathscr{L}_1$ is

$$\sigma_{\text{root}}(\mathscr{L}_1) = \max_{1 \leq a \leq n} \{\sigma_a + \Delta_a\}.$$

Step 1 (non-critical prefix). Because $\sigma_i - \sigma_{i-1} \geq d$ and $\Delta_a \leq d$, for every $a \leq i-1$ we have

$$\sigma_a + \Delta_a \leq \sigma_{i-1} + d \leq \sigma_i.$$

Hence, leaves $1, \ldots, i-1$ can never exceed the contribution of leaf i (whose term is at least σ_i). Therefore,

$$\sigma_{\text{root}}(\mathscr{L}_1) = \max_{a \geq i} \{\sigma_a + \Delta_a\}.$$

Intuitively, a sufficiently large gap at position i prevents earlier leaves from becoming critical in any depth-d implementation.

Step 2 (flatten the prefix). Define a modified input-MD vector

$$\mathscr{L}_3 = (\underbrace{\sigma_i, \ldots, \sigma_i}_{i \text{ entries}}, \sigma_{i+1}, \ldots, \sigma_n).$$

For any implementation (same Δ),

$$\sigma_{\text{root}}(\mathscr{L}_3) = \max\left\{\sigma_i, \max_{a \geq i}\{\sigma_a + \Delta_a\}\right\} = \max_{a \geq i}\{\sigma_a + \Delta_a\} = \sigma_{\text{root}}(\mathscr{L}_1).$$

Thus, *the ordering of implementations* (and hence, the set of optimizers) is identical under $\mathscr{L}_1$ and $\mathscr{L}_3$.

Step 3 (align to $\mathscr{L}_2$). Observe that $\mathscr{L}_2$ is obtained from $\mathscr{L}_3$ by subtracting the constant σ_i from every entry: the first i entries become 0, and the remaining entries become $\sigma_a - \sigma_i$. By Lemma 3.4.1 (Alignment), subtracting a constant from all inputs preserves the order of all candidate implementations (it shifts all $\sigma_a + \Delta_a$ by the same constant). Therefore, the set of optimizers under $\mathscr{L}_3$ and $\mathscr{L}_2$ coincides.

Algorithm 3.4: Deriving an input-MD signature

Input: Input MD $\mathscr{L} = (\sigma_1, \ldots, \sigma_n)$ (sorted); threshold θ.
Output: Signature *sig*.

```
sig ← L
// Alignment
foreach σ ∈ sig do
    σ ← σ − σ_1
// Dominance (detect first large gap)
for i ← n to 2 do
    if σ_i − σ_{i−1} ≥ θ then
        foreach σ ∈ sig do
            σ ← max{0, σ − σ_i}
        break
return sig
```

Combining the above steps shows that the optimal implementation for $\mathscr{C}_1$ (under $\mathscr{L}_1$) is also optimal for $\mathscr{C}_2$ (under $\mathscr{L}_2$). □

Dominance states that sufficiently large gaps in $\mathscr{L}$ exclude some leaves from any critical path; those leaves can be treated as if at MD 0.

The threshold θ depends on the cut size. For 5-cuts, $\theta = 3$ suffices because admissible fences have $d \leq 3$. Signatures are then used as cache keys together with the NPN representative:

Theorem 3.4.1. *Let $\mathscr{C}_1, \mathscr{C}_2$ be n-cuts with identical NPN representatives and input MDs $\mathscr{L}_1, \mathscr{L}_2$. If Algorithm 3.4 yields the same signature for $\mathscr{L}_1$ and $\mathscr{L}_2$, then $\mathscr{C}_1$ and $\mathscr{C}_2$ share the same FHE-cost-optimal implementation.*

Proof. Immediate from Lemmata 3.4.1 and 3.4.2. □

Summary. We classify queries by

(i) NPN representative of the local function, and

(ii) the input-MD signature.

This avoids redundant SAT calls and accelerates the global flow.

3.5 MC-aware MD Optimization

Building on Algorithm 3.3, we develop a scalable *MC-aware MD minimization* algorithm for leveled-FHE circuits. We then embed it in a two-stage optimization flow that interleaves our

Algorithm 3.5: MC-aware MD minimization (critical-path cut rewriting)

Input: XAG N implementing the target function.
Output: Optimized XAG N_{opt}.

$N_{\text{opt}} \leftarrow N$
cache ← precomputed optima for small functions (keyed by NPN rep and MD signature)
repeat
  $\Pi \leftarrow$ CRITICALPATH(N_{opt}) in topological order
  foreach *node* $n \in \Pi$ **do**
    $\mathbf{C}[n] \leftarrow$ ENUMERATECUTS(N_{opt}, n) // k-feasible cuts
    bestCut ← NULL, *bestLevel* ← $\delta(n)$
    foreach $\mathscr{C} \in \mathbf{C}[n]$ **do**
      $f \leftarrow$ local function of $\mathscr{C}$; $\mathscr{L} \leftarrow$ input MDs of $\mathscr{C}$
      $(f_r, sig) \leftarrow$ NPNCLASSIFY($f, \mathscr{L}$); $sig \leftarrow$ DERIVESIGNATURE(sig)
      $N_{\mathscr{C}} \leftarrow cache[(f_r, sig)]$
      if $N_{\mathscr{C}} =$ NULL **then**
        $N_{\mathscr{C}} \leftarrow$ EXACTSYNTHESISFORCUT($\mathscr{C}$) // Alg. 3.3
        $cache[(f_r, sig)] \leftarrow N_{\mathscr{C}}$
      $\delta_{\text{new}} \leftarrow$ LOCALMDAT($N_{\mathscr{C}}, n; \mathscr{L}$)
      if $\delta_{\text{new}} <$ *bestLevel* **then**
        bestLevel ← δ_{new}, *bestCut* ← $\mathscr{C}$, *bestImpl* ← $N_{\mathscr{C}}$
    if *bestCut* ≠ NULL **then**
      $N_{\text{opt}} \leftarrow$ REWRITE(*bestCut* → *bestImpl*, N_{opt})
until NOCRITICALMDIMPROVEMENT(N_{opt})
return N_{opt}

method with ESOP balancing [95], which is the strongest MD-only technique in the literature. The two methods explore largely orthogonal design spaces: our algorithm minimizes the root MD of critical cuts *subject to* MC-aware filters on admissible fences, whereas ESOP balancing aggressively reduces MD without regard to MC.

3.5.1 Algorithm Overview

High-level flow. Algorithm 3.5 iteratively walks the current critical path in topological order. For each node n, it enumerates k-feasible cuts rooted at n; for each cut $\mathscr{C}$, it looks up (or synthesizes) a locally optimal implementation under the current input MDs $\mathscr{L}$. Among candidates, it selects the one minimizing the node's local MD $\delta(n)$, then rewrites the network locally. The process repeats until no improvement in the critical-path MD is detected.

MC awareness. MC awareness enters in two complementary ways:

(i) the library of admissible fences (Section 3.3.3) excludes MD-reduction moves that would

cause disproportionate MC inflation; and

(ii) the exact synthesis for cuts (Algorithm 3.3) prioritizes schedules/fences by target root MD while respecting the admissible MC bound.

Caching. We cache solutions using two keys:

(i) the cut's NPN representative function (Section 3.4.5), and

(ii) the input-MD signature (Algorithm 3.4).

This eliminates redundant SAT calls across structurally different but equivalent synthesis queries.

Correctness note. For a fixed node n and cut $\mathscr{C}$, ExactSynthesisForCut returns a minimum-root-MD implementation under the admissible-fence set. Selecting the best $\mathscr{C}$ per node therefore yields a non-increasing sequence of critical-path MD values; termination occurs at a local optimum with respect to single-cut rewrites.

3.5.2 Leveled FHE Circuit Optimization Flow

ESOP balancing [95] and our MC-aware MD minimization attack MD from different angles. We therefore alternate them in a multi-start flow with a light exploration policy.

Passes. MCMD denotes Algorithm 3.5. ESOP calls ESOP balancing [95] over a moving window of cuts (as in Section 3.2.4). The driver alternates passes; if the selected pass fails to improve the objective, it switches to the other. If neither improves, the round terminates.

Restarts and relaxation. At each restart (except the first), we *relax* the structure to escape local minima by rewriting XOR as

$$a \oplus b = (\overline{a} \wedge b) \vee (a \wedge \overline{b}),$$

and implementing OR via De Morgan:

$$x \vee y = \overline{\overline{x} \wedge \overline{y}}.$$

This maps each XOR to three ANDs (plus negations), intentionally inflating MC/MD to expose new local-improvement opportunities for subsequent passes. The best-so-far circuit seeds the next restart.

Algorithm 3.6: FHE circuit optimization flow (leveled FHE)

Input: Initial XAG N.
Output: Best XAG N_{best} found.
Parameter : *num_restarts*, *cost_metric*.

$N_{\text{best}} \leftarrow N$
for $r \leftarrow 1$ **to** *num_restarts* **do**
  $N_{\text{cur}} \leftarrow (r = 1)\ ?\ N\ :\ \text{RELAXATION}(N_{\text{best}})$ `// escape local minima`
  while TRUE **do**
    $\eta \leftarrow \text{PICK}(\{\text{MCMD}, \text{ESOP}\})$ `// MCMD: Alg. 3.5; ESOP: [95]`
    $N' \leftarrow \text{APPLY}(\eta,\ N_{\text{cur}})$ **if** *cost_metric*$(N') \geq$ *cost_metric*(N_{cur}) **then**
      $\eta \leftarrow \text{SWITCH}(\eta)$
      $N'' \leftarrow \text{APPLY}(\eta,\ N_{\text{cur}})$
      if *cost_metric*$(N'') \geq$ *cost_metric*(N_{cur}) **then**
        break `// no progress by either pass`
      else
        $N_{\text{cur}} \leftarrow N''$
    else
      $N_{\text{cur}} \leftarrow N'$
  if *cost_metric*$(N_{\text{cur}}) <$ *cost_metric*(N_{best}) **then**
    $N_{\text{best}} \leftarrow N_{\text{cur}}$
return N_{best}

Objective. The flow is parameterized by *cost_metric*. In our experiments we consider both *MD*, which is the traditional objective in leveled-FHE circuit optimization, and our *leveled FHE cost* used throughout this chapter,

$$\text{FHE cost} = \text{MC} \times \text{MD}^2$$

(we omit "leveled" henceforth for brevity; cf. the sub-subsection on "Significant of MC-Aware MD Minimization" in Section 3.2.4). Swapping *cost_metric* lets us quantify the impact of the objective on end-to-end runtime.

Remarks on complementarity. ESOP balancing can unlock substantial MD reductions but may increase MC; our pass filters out MC-expensive moves by construction. Interleaving the two, therefore, exploits both kinds of opportunities: balanced ESOP trees expose low-MD cones, while MC-aware cut rewriting preserves parameter-friendly MC where it matters for leveled FHE.

3.6 Experimental Evaluation

We now present an empirical study of our methods, organized in two parts:

(i) MC-aware MD minimization as a standalone optimization algorithm; and

(ii) a full optimization flow that interleaves MC-aware MD minimization with ESOP balancing, evaluated under two objectives (MD and leveled FHE cost).

3.6.1 Experimental Setup

Platform. All experiments were run on an Apple M1 Max with 32 GB RAM.

Implementation. MC-aware MD minimization is implemented in C++ atop the `mockturtle` logic-network library. The exact-synthesis back end uses `bill` with `glucose` [13] as the SAT solver; both `mockturtle` and `bill` are part of the EPFL logic-synthesis libraries [165]. Homomorphic circuit evaluation uses `HElib` [93] with BGV in a binary plaintext space. We target $\lambda = 128$-bit security. Within the leveled-FHE setting, parameters are chosen by `HElib` so that the entire circuit can be evaluated without bootstrapping. To curb ciphertext growth, we conservatively relinearize after each homomorphic AND.

Baselines. We compare against `LOBSTER` [112], a state-of-the-art MD-reduction tool, and against ESOP balancing [95], the strongest MD-only reduction known. Because both ESOP balancing and our method are cut-based, we set the cut size to k=5 for all runs.

Benchmarks. Following [112], we evaluate the same benchmark suite for a fair comparison. A detailed introduction to the LOBSTER benchmark suite is available in Section 2.3.4.

3.6.2 Evaluating MC-aware MD Minimization

Table 3.2 reports MC, MD, optimization time ("Opt."), and homomorphic evaluation time ("Eval."). ESOP balancing and MC-aware MD minimization are each iterated to convergence; the number of passes is recorded ("# iter."). The fastest homomorphic evaluation per benchmark is highlighted in blue.

Optimization time. Because MC-aware MD minimization performs on-the-fly exact synthesis, it is typically slower to *optimize* than ESOP balancing; nevertheless the overhead is modest, owing to our SAT encoding, the cut-exact paradigm, and the cache-based query strategy that prevents redundant SAT calls. We could not time `LOBSTER` (no source), but [112] reports ~125 hours of rule learning *per benchmark* and ~8 hours of rewriting on large cases (e.g., `msort/isort/bsort`). Our method is four orders of magnitude faster in optimization time than `LOBSTER`.

Quality of the optimized circuits. Of the 25 benchmarks, at least one method improves 21. Among these, the new best implementations are attributed to `LOBSTER` (3), ESOP balancing (8), and MC-aware MD minimization (10). MC-aware MD minimization consistently yields

Table 3.2: MC-aware MD minimization vs. prior art.

Benchmark	Initial			LOBSTER			ESOP Balancing					MC-aware MD Minimization				
	MC	MD	Eval.[s]	MC	MD	Eval.[s]	MC	MD	# iter.	Opt.[s]	Eval.[s]	MC	MD	# iter.	Opt.[s]	Eval.[s]
cardio	109	10	14.47	116	8	10.00	120	8	3	0.11	10.37	108	8	3	9.81	**9.35**
dsort	708	9	59.91	793	8	51.84	948	7	2	0.13	50.49	708	8	2	3.22	**45.18**
msort	810	45	1 964.69	1 450	36	**1 125.30**	1 569	36	8	0.75	1 229.30	774	45	2	0.74	1 228.88
isort	810	45	1 830.12	1 482	36	**1 069.88**	1 569	36	8	0.75	1 227.46	774	45	2	0.49	1 212.08
bsort	810	45	1 900.51	1 482	36	**1 112.19**	1 569	36	8	0.75	1 226.71	774	45	2	0.49	1 213.17
osort	702	25	520.77	1 404	20	333.97	1 404	20	6	0.47	405.50	638	25	2	0.62	**273.56**
hd01	87	6	5.16	87	6	5.52	102	5	2	0.00	**3.42**	87	6	1	0.27	5.53
hd02	76	6	5.67	76	6	5.24	76	6	1	0.00	5.15	76	6	1	0.27	5.13
hd03	27	5	1.78	27	5	1.38	30	4	2	0.02	1.45	29	4	2	0.89	1.43
hd04	75	10	20.79	78	8	9.20	74	7	3	0.05	**5.09**	63	8	4	9.53	6.03
hd05	121	7	9.15	121	7	8.02	184	6	2	0.01	10.00	121	7	1	0.09	7.95
hd06	121	7	7.11	121	7	8.00	184	6	2	0.02	10.02	121	7	1	0.06	8.00
hd07	17	5	4.25	13	3	0.99	19	3	2	0.01	0.51	15	3	3	0.78	**0.48**
hd08	18	6	2.03	18	5	1.00	26	4	2	0.01	1.24	16	5	2	0.17	**0.94**
hd09	134	14	30.42	177	10	20.73	173	10	4	0.06	17.60	134	10	4	8.78	**14.78**
hd10	35	6	3.54	36	5	1.71	34	5	1	0.01	1.69	32	5	4	1.51	**1.66**
hd11	391	18	133.65	391	15	93.54	411	13	2	0.08	**79.46**	385	14	3	9.74	83.46
hd12	116	16	29.73	116	15	26.53	126	12	3	0.09	**20.25**	107	13	3	0.76	24.22
bar	3 141	12	460.61	3 015	11	370.77	2 266	8	2	0.15	163.12	1 841	9	4	28.85	**149.08**
cavlc	655	16	144.26	668	10	63.88	713	8	7	0.37	**52.67**	607	11	7	42.89	76.46
ctrl	107	8	9.87	120	5	4.15	107	4	5	0.03	3.97	89	4	5	9.85	**3.49**
dec	304	3	4.03	304	3	4.08	304	3	1	0.01	4.13	292	3	2	0.32	**3.97**
i2c	1 157	15	311.52	1 215	8	93.34	1 254	7	9	0.24	**71.53**	1 152	10	6	19.95	109.30
int2float	213	15	56.27	234	8	17.43	240	7	5	0.10	**14.14**	205	10	5	4.34	21.21
router	170	19	97.48	190	10	30.18	232	9	5	0.13	**19.34**	186	13	4	1.95	37.28
Total			7 627.78			4 468.87					4 634.61					4 542.62
Norm.			1.71			1.00					1.04					1.02

Notes. "Opt.[s]" is optimization time; values < 0.005 s are shown as 0.00 s. "Eval.[s]" is homomorphic circuit evaluation time in `HElib`.

the lowest MC among the three — by design, it trades MD reductions only when MC remains controlled — while ESOP balancing typically attains the lowest (or tied) MD compared with LOBSTER.

When MC-aware MD minimization underperforms. On a few instances where LOBSTER or ESOP balancing wins, our method may converge in fewer iterations and avoid transient high-MC regions that could ultimately lead to lower FHE cost. This observation motivates the hybrid flow in Section 3.5.2: ESOP balancing can aggressively reduce MD without MC constraints, after which MC-aware MD minimization consolidates gains under an MC-aware envelop.

3.6.3 Evaluating the Full Optimization Flow

We evaluate the flow of Algorithm 3.6 under two objectives: *FHE-cost-oriented* (objective = $\text{MC} \times \text{MD}^2$) and *MD-oriented*. We run five restarts to balance runtime and search breadth. For each benchmark we highlight in blue the fastest homomorphic evaluation time across all configurations (including Table 3.2); ties on MC/MD are all highlighted even if the evaluation latency differs slightly.

The flow (either objective) delivers the best-known implementations on 17 of the 21 improvable benchmarks. Notably, 11 designs were *not* reachable by any single method (LOBSTER,

Table 3.3: Effect of the flow's objective: leveled FHE cost vs. MD.

Benchmark	FHE-cost-oriented				MD-oriented			
	MC	MD	Opt.[s]	Eval.[s]	MC	MD	Opt.[s]	Eval.[s]
cardio	108	8	70.36	**9.35**	117	8	45.61	10.16
dsort	708	7	26.77	**39.42**	948	7	38.28	50.50
msort	788	42	53.86	**881.38**	1 391	36	182.86	1 180.46
isort	816	42	45.56	**915.46**	1 324	36	88.24	1 178.53
bsort	788	42	45.45	**880.41**	1 354	36	107.31	1 214.01
osort	750	24	20.54	320.54	1 261	20	121.63	394.29
hd01	102	5	0.14	**3.42**	102	5	0.22	**3.38**
hd02	76	6	3.53	5.13	76	6	1.89	5.15
hd03	30	4	3.88	1.27	30	4	1.46	1.49
hd04	67	7	17.38	**4.81**	74	7	14.05	5.11
hd05	121	7	7.95	8.08	184	6	5.66	10.03
hd06	121	7	5.18	7.99	184	6	7.06	10.00
hd07	15	3	1.02	**0.48**	15	3	0.65	**0.49**
hd08	21	4	0.67	**0.92**	21	4	0.62	**1.12**
hd09	155	10	14.84	16.38	150	10	11.46	16.00
hd10	32	5	1.44	**1.67**	32	5	0.55	**1.71**
hd11	423	13	32.12	83.88	410	13	22.90	80.88
hd12	115	12	3.73	**19.35**	115	12	3.96	**19.30**
bar	1 942	8	127.06	**145.10**	2 710	7	185.75	160.52
cavlc	691	9	122.10	56.19	717	8	85.62	53.05
ctrl	97	4	10.76	3.71	115	3	12.94	**2.02**
dec	292	3	1.81	**3.96**	292	3	2.28	**3.94**
i2c	1 252	7	57.85	**70.74**	1 236	8	22.80	91.04
int2float	217	8	32.24	17.24	309	6	10.49	17.10
router	229	9	19.38	**19.06**	257	9	15.03	22.07
Total				3 515.94				4 532.35
Norm.				0.79				1.01

Notes. We use five restarts. Total homomorphic circuit evaluation times are normalized to LOBSTER.

ESOP balancing, or MC-aware MD minimization) alone, underscoring the complementarity of the passes. On msort, isort, and bsort — previously dominated by LOBSTER — our FHE-cost-oriented flow improves homomorphic evaluation time by 21.68%, 14.43%, and 20.84%, respectively, versus LOBSTER. Under the MD-oriented setting, the flow also discovers new lowest-MD designs for bar, ctrl, and int2float.

Impact of the objective. With leveled FHE cost as the objective, the flow attains the best solutions on 15 of the 21 improved benchmarks and reduces total homomorphic evaluation time by 21.32% relative to LOBSTER; compared to the *initial* designs, the reduction is 53.91% (more than 2× speedup). Using MD as the objective yields the best solutions on 7 benchmarks and a total homomorphic evaluation time slightly worse than LOBSTER. Despite often achieving the smallest MD, MD-oriented runs can underperform in Eval. due to elevated MC — highlighting the practical importance of MC-aware objectives in leveled FHE.

3.7 Discussion

This section reflects on the capabilities, limitations, and future potential of the proposed MC-aware MD–minimization framework for Boolean leveled-FHE circuit synthesis. While

our method improves homomorphic circuit evaluation by addressing *both multiplicative complexity* (MC) and *multiplicative depth* (MD), it also opens several directions for further research. We first articulate why joint optimization matters, then discuss extensibility to arithmetic circuits, and finally outline limitations — most notably in cost modeling — together with promising avenues for future work.

3.7.1 The Importance of Joint MC–MD Optimization

Our experiments demonstrate that MD-only optimization can degrade homomorphic circuit evaluation when it inflates MC. This confirms that MC and MD jointly determine leveled-FHE performance and motivates optimization frameworks that explicitly account for both. The contribution of this chapter is therefore twofold:

(i) an algorithmic strategy that *concurrently* reasons about MC and MD at the cut level, and

(ii) a *reconfigurable* objective interface.

Although we instantiate the objective with the empirically grounded leveled FHE cost MC × MD^2, the optimization infrastructure accepts any user-specified $\kappa(MC, MD)$. This design allows practitioners to adopt scheme- or implementation-specific objectives and supports future, data-driven calibration of cost models.

3.7.2 Extension to Arithmetic FHE Circuits

Arithmetic-circuit optimization is especially pertinent in the *leveled* setting, where schemes such as BGV/BFV (and CKKS for approximate arithmetic) are widely regarded as more efficient than fast-bootstrapping schemes when the plaintext modulus or vector width is large. In this regime, performing word-level operations directly can amortize costs across many bits/slots and avoid the overheads associated with frequent bootstrapping.

We have conducted preliminary trials toward this goal using *e-graphs* (*equality-saturation*) [167] to systematically apply algebraic rewrites that do not rely on Boolean-specific properties — e.g., commutativity, associativity, and distributivity — thus operating over rings/fields beyond $\mathbb{B}$ [46]. This machinery is well-suited to arithmetic FHE circuits, where rich algebraic identities can substantially reduce ciphertext multiplications and improve data layout before low-level synthesis.

While the present chapter targets Boolean circuits, the methodology admits a natural generalization to arithmetic datapaths. The exact-synthesis core currently verifies functionality via SAT over $\mathbb{B}$ (Section 3.3.2); for arithmetic circuits, one can replace SAT with *satisfiability modulo theories* (SMT) over fixed-size bit-vectors or finite rings/fields, preserving the overall search structure:

1. enumerate admissible “AND fences” (generalized to arithmetic multiplicative levels),
2. schedule inputs under level constraints,
3. solve the resulting satisfiability instances to witness feasibility.

In this setting, MC counts ciphertext multiplications and MD tracks multiplicative levels at word granularity.

In short, although our current implementation does not *directly* optimize arithmetic FHE circuits, it provides a principled scaffold that can be extended with SMT-based equivalence, e-graph–driven algebraic rewriting, and level-aware cost modeling to form a powerful tool for low-cost arithmetic synthesis in leveled FHE.

3.7.3 Limitations and Directions for Future Work

Level-aware costs. Our experiments suggest that a uniform multiplication cost can be an oversimplification. In leveled FHE, each multiplication consumes a modulus “layer”; as depth increases, remaining layers decrease and subsequent multiplications can become cheaper due to smaller ciphertext moduli and keys. This effect — visible in benchmarks such as `msort/isort/bsort` — indicates that multiplication cost should depend on *where* it occurs (its multiplicative level), not only on *how many* occur in total. Incorporating level-sensitive costs (e.g., weighting multiplicative layers or learning a per-level cost profile from measurements) is a compelling next step.

Operations beyond XOR/AND. Our Boolean model intentionally abstracts away operations such as rotations, key switching, and relinearization scheduling (beyond the conservative rule used here), which can materially affect runtime in application compilers. Integrating such costs into the cost model $\kappa(\cdot)$, or augmenting the fence/schedule space to include them, could further tighten the link between the objective and homomorphic circuit evaluation.

Scalability and search. Exact cut synthesis is effective because it reuses solutions via NPN classification and MD-signatures, but further scalability gains are possible: larger cut sizes with incremental SAT/SMT; proof logging to cache UNSAT cores across related queries; and parallel cut solving. These engineering directions are orthogonal to the algorithmic core.

In summary, this chapter introduces, to our knowledge, the first framework that *jointly* optimizes MC and MD for leveled-FHE circuit synthesis. Beyond advancing the state of the art in homomorphic circuit evaluation, the approach contributes a general, pluggable optimization substrate on which richer, level-aware and operation-aware cost models can be deployed as they mature.

4 Technology Mapping for TFHE: Gate-Set Design and Multi-Value PBS

4.1 Motivation

From depth-driven to bootstrap-driven optimization. Chapter 3 showed that, in leveled schemes, homomorphic cost is governed by *multiplicative depth* (MD) together with the number of multiplications (MC), motivating MC–MD co-optimization. In fast-bootstrapping schemes such as TFHE [57], the performance determinant shifts: each Boolean gate triggers a *programmable bootstrapping* (PBS), which both evaluates a function and refreshes noise. As a result, homomorphic circuit evaluation correlates primarily with the *number of PBS operations*, rather than with MD. This chapter, therefore, targets a synthesis objective orthogonal to Chapter 3: *minimize PBS operations*, under a TFHE-native cost model.

Gaps in current compiler practice. In recent years, we have witnessed many remarkable TFHE compilers that translate Boolean functions, described in high-level programs such as *hardware description language* (HDL) as adopted by `Cingulata` [42], `Romeo` [89], or C++ as adopted by Google's `Transpiler` [86], into homomorphic Boolean circuits, with each gate bound to certain homomorphic computation implemented in the `TFHE`[1] library. These compilers enable non-experts to develop secure and efficient FHE applications, significantly facilitating the transition of recent developments in FHE schemes from theory to practice. However, we observe that the exploration of circuit optimization — an essential task in a compilation process that strongly affects the resulting homomorphic evaluation efficiency — remains insufficient in these compiler designs.

The evaluation of each homomorphic Boolean gate involves performing a PBS operation, the most computationally intensive task in TFHE. Consequently, the efficiency of homomorphically evaluating a Boolean function is closely correlated with the gate count of the circuit design. This correlation frames TFHE circuit [2] synthesis as a general area-oriented Boolean circuit synthesis problem. It motivates existing TFHE compilers to rely on off-the-shelf *elec-*

[1] To avoid confusion, we distinguish between the TFHE scheme and the `TFHE` library using different fonts.
[2] In this chapter, we use "TFHE circuit" and "homomorphic Boolean circuit" interchangeably.

tronic design automation (EDA) tools, originally developed for hardware circuit synthesis, to achieve compact TFHE circuit designs. For instance, `Romeo` leverages the open-source technology mapper `Yosys` [186] to map HDL designs to two-input Boolean gates and three-input multiplexers supported by the `TFHE` library [89]. A similar methodology is also adopted by `Transpiler` [86]. However, this method has two major limitations:

(i) *Gate-set myopia.* It is constrained by the gate set supported by the original `TFHE` library. Existing literature indicates that larger fan-in Boolean gates can also be exploited to deliver more compact homomorphic Boolean circuit designs [31, 123], yet effectively exploiting them remains an open problem.

(ii) *Insufficient exploitation of emerging cryptographic techniques.* Specifically, the *multi-value PBS* technique [43] suggests that, for multiple Boolean gates with the same supports (i.e., inputs), their homomorphic evaluation can share a single FBS operation. From a circuit synthesis perspective, this implies that the cost of a multi-output Boolean gate can be close to that of a single-output gate, presenting an opportunity to synthesize higher-performance TFHE circuits. However, this opportunity has not been thoroughly investigated by existing TFHE compiler designs.

Problem statement. Given an arbitrary Boolean function, synthesize a TFHE circuit that:

(1) selects an appropriate library of TFHE-realizable gates,

(2) maps the network to *minimize the number of PBS operations* (proxy for homomorphic cost), and

(3) *coalesces gates with identical supports* to exploit multi-value PBS.

Unlike the leveled setting, MD is immaterial here (noise is refreshed per PBS); the core challenge is to reduce and share PBS operations.

This chapter seeks to address the aforementioned limitations. We approach the TFHE circuit synthesis problem as a technology mapping issue. The approach is not limited to a subset of functions and applies to any Boolean function. To that end, we first analyze the intrinsic properties of large-fan-in Boolean gates to design a suitable gate set for homomorphic Boolean circuit synthesis (Section 4.3.1). We then customize a multi-value-FBS-aware and area-oriented mapping algorithm to maximally enhance circuit quality and ultimately reduce the computational overhead of homomorphic evaluation (Sections 4.3.2–4.4).

Scope and positioning. This chapter operates in the *single-ciphertext-space* TFHE regime, exploiting TFHE-native expressiveness — i.e., the size/complexity of a Boolean function realizable by a single PBS — to *minimize the number of PBS invocations.* Chapter 5 generalizes this

to *dual-ciphertext-space* designs that strategically switch spaces to leverage complementary properties and further improve homomorphic evaluation efficiency. Together, these two chapters constitute the TFHE strand of Part A and complement the leveled-scheme optimization developed in Chapter 3.

Approach preview. In this chapter, we propose casting TFHE circuit synthesis as *technology mapping*. First, we analyze and build a TFHE-relevant gate library that captures the cost/benefit of large-fan-in functions under PBS (Section 4.3.1). Second, we develop a *multi-value-PBS–aware, area-oriented* mapper that

(i) prioritizes gates that share support (Section 4.3.2), and

(ii) identifies and coalesces them to multi-output gates, to facilitate multi-value-PBS-based evaluation (Section 4.4).

This design is *function-complete* (handles arbitrary Boolean functions) and compiler-friendly: it can serve as a specialized back end beneath existing front ends.

Outcomes. As demonstrated in Section 4.5, this TFHE-native mapping reduces synthesis time and, more importantly, lowers homomorphic evaluation cost by reducing and sharing PBS operations — achieving substantial speedups over prior TFHE compilation flows — while remaining broadly applicable to fast-bootstrapping schemes beyond TFHE.

This chapter builds upon the results originally presented in [196].

4.2 Preliminaries

This section reviews the facets most relevant to TFHE-based synthesis. We briefly position modern FHE families (Section 4.2.1), summarize TFHE and its PBS primitive (Sections 4.2.2), recall the homomorphic Boolean gate notions used by our design (Sections 4.2.3), and survey related work in TFHE-oriented synthesis (Section 4.2.4).

4.2.1 Fully Homomorphic Encryption

FHE enables computation over encrypted data without decryption. Modern schemes split into two mainstream branches:

Leveled schemes (e.g., BGV [35] and BFV [81]) support a bounded number of homomorphic multiplications by choosing parameters large enough to accommodate a target multiplicative depth. Bootstrapping is optional and typically expensive; see Chapter 3.

Fast-bootstrapping schemes refresh noise *frequently*. The bootstrapping operation in these schemes is realized significantly differently from leveled schemes and is commonly referred to as *programmable bootstrapping* (PBS). A PBS operation not only refreshes ciphertexts but also evaluates an operation simultaneously. The most recent advancement in the fast-bootstrapping branch is the *torus FHE* (TFHE) scheme [57], a successor of the *GSW* scheme [83] and the *FHEW* scheme [75], which provides efficient evaluation of Boolean functions.

4.2.2 Torus FHE

TFHE is a state-of-the-art FHE scheme optimized for fast and efficient evaluation of Boolean functions. TFHE achieves this through an innovative bootstrapping mechanism that maintains low noise levels and high performance. Messages are encrypted as `LWE` samples or ciphertexts, one message per sample. With the plaintext space size denoted as p, the `LWE` encryption of message $m \in \mathbb{Z}_p$ is a tuple $(\vec{a}, b)$ such that $b - \vec{a} \cdot \vec{s} \approx m\frac{1}{p}$.

Programmable Bootstrapping

The bootstrapping operation in fast-bootstrapping schemes, often called *programmable bootstrapping* (PBS), evaluates any function $F : \mathbb{Z}_p \mapsto \mathbb{Z}_p$ while refreshing noise. Conceptually, the procedure can be split into four steps:

1. **Phase discretization and group embedding.** Given an input `LWE` ciphertext $c = (\vec{a}, b)$, the coefficients are rounded (or modulus–switched) to $\mathbb{Z}_{2N}$. We use a cyclic multiplicative group $\mathcal{G} \cong \mathbb{Z}_{2N}$ to represent these residues, realized as the powers of X in the quotient ring modulo the power-of-two cyclotomic polynomial:

   $$\mathcal{G} = \{X^k \bmod (X^N + 1) \mid k \in \mathbb{Z}_{2N}\}.$$

2. **Linear combination.** We homomorphically evaluate the linear part of `LWE` decryption to obtain an `RLWE` encryption of X^φ, where the discretized phase

   $$\varphi \approx b - \vec{a} \cdot \vec{s} \in \mathbb{Z}_{2N}.$$

   This is achieved using the bootstrapping key (i.e., `GSW` encryptions of $\{X^{s_i}\}_i$). The result is the *accumulator*

   $$\mathsf{ACC} = \mathsf{Enc}_{\mathtt{RLWE}}\left(X^\varphi\right) \in \mathcal{G}.$$

3. **Blind rotation.** We multiply the accumulator by a test polynomial (test vector) TV_G that encodes a function $G : \mathbb{Z}_{2N} \mapsto \mathbb{Z}_p$. We take

   $$G = F \circ r_p, \qquad r_p : \mathbb{Z}_{2N} \mapsto \mathbb{Z}_p,$$

where r_p is the *rounding map* that recovers the (noise-free) message $m = r_p(\varphi)$ from the noisy phase, and $F : \mathbb{Z}_p \mapsto \mathbb{Z}_p$ is the desired payload function. The product $\mathsf{TV}_G \cdot \mathsf{ACC}$ implements this lookup homomorphically — thus commonly referred to as the *blind rotation* step — and produces an RLWE encryption of $G(\varphi) = F(w)$.

4. **Sample extraction.** Finally, we extract an LWE ciphertext from the RLWE ciphertext $\mathsf{TV}_G \cdot \mathsf{ACC}$, yielding an LWE encryption of $F(w)$ (equivalently, $G(\varphi)$).

4.2.3 Homomorphic Logic Gate Evaluation via PBS

To evaluate f via a PBS operation, we first map each input pattern $\vec{b} = (b_1, \ldots, b_n) \in \mathbb{B}^n$ to an *index* in a small plaintext ring by means of a *projection (indexing) function* (Step 2)

$$\phi : \mathbb{B}^n \mapsto \mathbb{Z}_p.$$

Operationally, given LWE encryptions of the bits b_i, the index $\phi(\vec{b})$ is formed *homomorphically* by a weighted linear combination:

$$\mathsf{Enc}_{\mathtt{LWE}}(b_1), \ldots, \mathsf{Enc}_{\mathtt{LWE}}(b_n) \overset{\text{lin. comb.}}{\Longrightarrow} \mathsf{Enc}_{\mathtt{LWE}}\left(\textstyle\sum_{i=1}^{n} w_i \cdot b_i \bmod p\right) = \mathsf{Enc}_{\mathtt{LWE}}\big(\phi(\vec{b})\big), \tag{4.1}$$

where $w_i \in \mathbb{Z}_p$ are compiler-chosen weights. The subsequent blind rotation interprets this encrypted index as a rotation exponent and steers the accumulator accordingly.

To return the correct gate output from the encrypted index, we encode a *look-up table* (LUT) $F : \mathbb{Z}_p \mapsto \mathbb{B}$ so that, for all input patterns,

$$f(b_1, \ldots, b_n) = F\big(\phi(b_1, \ldots, b_n)\big).$$

This correctness relation ensures that querying the LUT at the (encrypted) index produced by ϕ, yielding an LWE encryption of the desired output bit.

LUT Construction in the Negacyclic Ring

The LUT F is embedded in the degree-N ring $\mathbb{T}[X]/(X^N + 1)$, where $\mathbb{T}$ is the discretized torus. Because $X^N = -1$ in this quotient, the ring is *negacyclic* and admits $2N$ coefficient slots arranged in pairs related by negation. In practice, we choose p so that $p \mid 2N$, and we replicate each LUT entry across $\frac{2N}{p}$ equally spaced coefficients to

(i) align with the blind-rotation orbits, and

(ii) distribute noise uniformly.

Negacyclicity imposes a symmetry: entries in the second half of the ring are determined (up to sign) by those in the first half. Writing indices modulo $2N$ and letting the freely chosen half

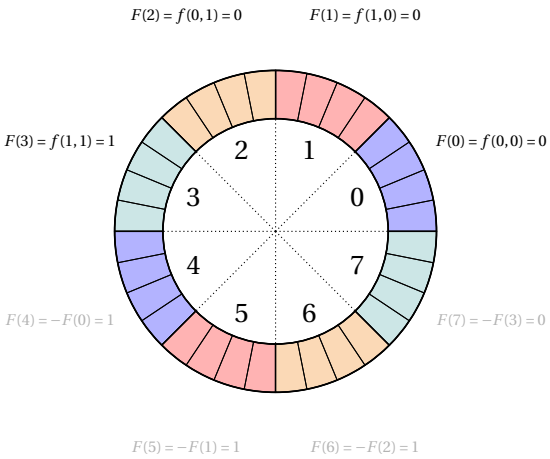


Figure 4.1: Naïve LUT encoding for a two-input AND gate using $\phi(b_1, b_2) = b_1 + 2b_2$ with plaintext space $\mathbb{Z}_8$ ($p = 8$) and polynomial degree $N = 16$. Colored segments show distinct LUT entries; each pair of freely assigned and negacyclically determined (negated) entries shares a color. Because ϕ ranges only over $\{0, 1, 2, 3\}$, all queried locations lie in the freely assignable half of the ring. Unqueried locations are "don't-care" for correctness.

be $0, \ldots, \frac{p}{2} - 1$, one must satisfy

$$F(w) = -F\left(w - \tfrac{p}{2}\right) \quad \text{for all } w \in \left\{\frac{p}{2}, \ldots, p-1\right\}. \tag{4.2}$$

Thus, only $p/2$ LUT values are independent (namely, not subject to the replication effect). The blind rotation lands on the coefficient block corresponding to the encrypted index (Step 3), and the associated value is then extracted from the test vector (Step 4).

Naïve (Uncompressed) Indexing

When no truth table compression is applied, a natural choice is the radix-2 map

$$\phi(b_1, \ldots, b_n) = \sum_{i=1}^{n} 2^{i-1} \cdot b_i \ \in \mathbb{Z}_{\frac{p}{2}}, \tag{4.3}$$

which assigns a unique index to each input pattern. This requires $p \geq 2^n$ (and p is typically a power of two with $p \mid 2N$) so that all patterns are addressable.

Figure 4.1 illustrates this placement for a two-input AND gate with $\phi(b_1, b_2) = b_1 + 2b_2$ in $\mathbb{Z}_8$. With $N = 16$, each LUT entry is replicated across $\frac{2N}{p} = 4$ coefficients at equal angular spacing. Since the maximum index produced by ϕ is 3, all accesses fall within the first (freely encodable) half of the ring; the negacyclic counterparts are fixed by sign, and the remaining indices are never queried during evaluation and may be treated as *don't-care* values. This explicit embedding ties directly to the four-step PBS procedure above: the linear combination produces the encrypted index, the blind rotation aligns the accumulator to the correct block,

and the test polynomial effects the lookup before sample extraction.

Multi-Value Programmable Bootstrapping

Multi-value programmable bootstrapping (MV-PBS) is a key optimization that significantly improves throughput by evaluating *several* functions on the *same* encrypted input within a *single* PBS pass [43]. Concretely, suppose one wishes to compute an n-input, m-output Boolean operator $f : \mathbb{B}^n \mapsto \mathbb{B}^m$, whose m coordinate functions share the same inputs and the same projection weights (cf. Eq. (4.1) and the sub-section on "LUT construction in the negacyclic ring"). In the standard approach, one would invoke PBS m times — once per output. In the multi-value variant, the most expensive component of PBS, the *blind rotation*, is executed *once* and its result (the accumulator) is reused to derive all m outputs.

At a high level, this reuse is enabled by factoring the per-output test polynomials into a *common* factor and a small, *output-specific* factor. The common factor drives a single blind rotation, yielding a shared accumulator ACC that encodes the appropriately rotated state for the encrypted index. Each output is then produced by multiplying ACC by its output-specific factor, followed by sample extraction. Because these final multiplications are substantially less costly than additional blind rotations, the total cost becomes dominated by the single rotation, and the marginal cost per extra output is small.

Careful selection of low-norm test polynomials together with optimized polynomial arithmetic keeps noise growth under control, preserving decryption correctness. Benchmarks in [43] report ~ 1.6s to evaluate a six-input, m-output Boolean gate (for large m), with runtime nearly independent of m, thanks to blind-rotation amortization. While results for smaller fan-in were not reported, the cost model suggests a super-linear speedup as fan-in decreases, since ring dimension and test-polynomial degrees can be reduced accordingly.

4.2.4 Related Works

We review prior work along two orthogonal axes that motivate our synthesis formulations: support for *large-fan-in* Boolean gates, and *multi-output*-aware mapping that exploits MV-PBS.

Large-fan-in Boolean gates

When a Boolean computation is evaluated homomorphically under TFHE, each gate typically triggers a PBS; hence, reducing the gate count directly reduces the number of PBS operations. From a circuit synthesis viewpoint, TFHE circuit construction can be framed as area-oriented logic synthesis with *unit* gate cost, and technology mapping is used to target a specific gate library. Existing compilers such as `Romeo` [89] and Google's `Transpiler` [86] leverage general-purpose tools (e.g., `Yosys` [186]) to map to the gate set implemented by the `TFHE` library, which

consists of two-input gates and three-input multiplexers).

However, several studies indicate that some larger cells (e.g., full adders [123]) can be evaluated homomorphically with cost comparable to the small gates provided by the library, enabling more compact designs. This motivates expanding the target set beyond the default library. Two strategies have emerged:

1. *Fixed, limited extensions.* One augments the TFHE gate set with a small number of larger gates (e.g., only full adders [133]) and then applies standard technology mapping. This approach is simple but under-exploits the design space due to the narrow extension.

2. *Local compounding of small gates.* Starting from a two-input network, one locally *compounds* chains of small gates into larger ones and validates the compound on the fly. Greedy compounding (as in [90]) maximizes local use of large gates but lacks a global objective and can miss globally compact solutions; [40] mitigates this by deferring some decisions (keeping several local candidates), yet the approach remains inherently local. These observations suggest that *global* technology mapping is a more principled way to exploit large-fan-in expressiveness in TFHE.

Multi-Output Boolean Gates

MV-PBS (cf. the sub-subsection on "multi-value programmable bootstrapping" in Section 4.2.3) adds a powerful dimension: the cost of an n-input, m-output gate can be *close* to that of the corresponding single-output gate, provided all outputs use the same input set and projection weights. This strongly favors *multi-output* mapping when such structure is present.

Existing work has only scratched the surface of this topic. To the best of our knowledge, current compilers exploit MV-PBS via a post-synthesis *merging* pass that scans a mapped netlist and groups eligible single-output gates into multi-output ones whenever possible [133, 90]. While beneficial, this after-the-fact merging is agnostic to upstream structural choices that could create substantially more (or better aligned) sharing opportunities.

Directly transplanting hardware-centric algorithms is also limiting, because near-unit-cost multi-output cells are rare in silicon flows. The closest analogues — dual-output LUTs for *field programmable gate arrays* (FPGAs) [180] and multi-output standard cells such as half/full adders [168] — support at most two outputs and do not capture the TFHE setting where the amortization improves with *larger m*. In contrast, our TFHE context routinely benefits from 3-5 outputs per gate (and potentially more), which calls for mapping algorithms that plan for multi-output structure, rather than opportunistically merging at the end. This gap motivates the multi-output–aware technology mapping and cost models developed in this chapter.

4.3 TFHE Circuit Synthesis via Technology Mapping

Logic gates implementing selected Boolean functions — with fan-in possibly larger than two — can be used to synthesize TFHE circuits under standard security parameters. By packing more logic into a single gate invocation, larger-fan-in cells reduce the number of PBS operations and yield more compact netlists, thereby improving homomorphic circuit evaluation efficiency without compromising security.

This section addresses two questions:

1. Which larger-fan-in gates are admissible in the TFHE setting (Section 4.3.1)?
2. How can we map arbitrary logic to such a gate set while minimizing gate count (Section 4.3.2)?

4.3.1 Homomorphic Gate Set

In TFHE, a gate's truth table is encoded into the $\frac{p}{2}$ "slots" of a test polynomial used by PBS (cf. the sub-subsection on "programmable bootstrapping" in Section 4.2.2). For an n-input gate, the truth table has 2^n entries. Note that once the scheme parameters are fixed, the plaintext space $\mathbb{Z}_p$ is fixed. Thus, with the naïve encoding in Eq. 4.3, an n-input gate is admissible only if the scheme parameters are configured in a way that the plaintext size $p \geq 2^{n+1}$, in order to fit the 2^n-entry table into the $\frac{p}{2}$ encoded entries. Otherwise, to improve the homomorphic evaluation efficiency by choosing a smaller p and facilitating relaxing the parameter selection, it is necessary to *compress* the table entries.

Compression identifies multiple input patterns with a single encoded slot. In TFHE, this is enabled by two mechanisms:

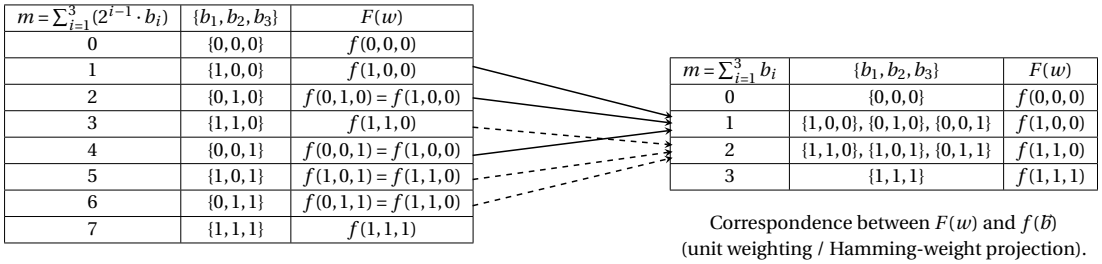
(i) a tailored *projection* (weighting, Eq (4.1)) $\phi(b) = \sum_i w_i \cdot b_i \bmod p$ so that many patterns sharing the same output collapse to the same index; and

(ii) the *negacyclic* constraint of the ring encoding (cf. the sub-subsection on "LUT construction in the negacyclic ring" in Section 4.2.3), which halves the degrees of freedom by relating the second half of indices to the first.

Below, we characterize admissible three-input gates for the common setting $p = 8$, rather than the original requirement that $p = 16$ (2^{3+1}), and use this to define a practical TFHE gate library.

Symmetric Gates

If a gate is *symmetric* (its output depends only on the Hamming weight of the input), assigning *equal* weights to all inputs collapses the 2^n entries onto at most $n+1$ distinct indices — the possible Hamming weights. For $n = 3$ and $p = 8$, this fits comfortably.



$m=\sum_{i=1}^{3}(2^{i-1}\cdot b_i)$	$\{b_1,b_2,b_3\}$	$F(w)$
0	{0,0,0}	$f(0,0,0)$
1	{1,0,0}	$f(1,0,0)$
2	{0,1,0}	$f(0,1,0)=f(1,0,0)$
3	{1,1,0}	$f(1,1,0)$
4	{0,0,1}	$f(0,0,1)=f(1,0,0)$
5	{1,0,1}	$f(1,0,1)=f(1,1,0)$
6	{0,1,1}	$f(0,1,1)=f(1,1,0)$
7	{1,1,1}	$f(1,1,1)$

Correspondence between $F(w)$ and $f(\vec{b})$ (default radix-2 weighting).

$m=\sum_{i=1}^{3} b_i$	$\{b_1,b_2,b_3\}$	$F(w)$
0	{0,0,0}	$f(0,0,0)$
1	{1,0,0}, {0,1,0}, {0,0,1}	$f(1,0,0)$
2	{1,1,0}, {1,0,1}, {0,1,1}	$f(1,1,0)$
3	{1,1,1}	$f(1,1,1)$

Correspondence between $F(w)$ and $f(\vec{b})$ (unit weighting / Hamming-weight projection).

Figure 4.2: Compression for three-input symmetric gates. Left — default radix-2 weights yield a distinct index per input pattern. Right — unit weights ($\phi=\sum_i x_i$) collapse inputs by Hamming weight, reducing the LUT to four indices. Arrows indicate many-to-one projection from the uncompressed to the compressed table.

Projection choice. For symmetric gates we use *unit* weights; the PBS index is

$$\phi_{\mathrm{sym}}(b_0,\ldots,b_{n-1})=\sum_{i=0}^{n-1} b_i,$$

so that inputs with the same Hamming weight collide to the same ϕ and their truth-table entries are compressed. Moreover, using unit weights keeps the largest index small, which helps maintain low noise growth in PBS.

Conclusion. All three-input symmetric gates are admissible under $p=8$ with a uniform weighting.

Negacyclic Gates

Because the test polynomial lives in the negacyclic ring $\mathbb{T}[X]/(X^N+1)$, indices in the second half are tied to those in the first by $X^N=-1$. Due to the negacyclic property, if $\exists\, k\in[2^{n-1}..2^n)$ such that $\forall\, i\in[k..2^n): f_i=\neg f_{i-k}$[3], then, only the first k entries must be encoded; the remaining entries are determined by negacyclic symmetry. For $n=3$, admissibility under $p=4$ requires $k=2^{n-1}=4$, i.e., the second half of the table is the bitwise negation of the first.

Definition 4.3.1 (Negacyclic Boolean function)**.** *A Boolean function $f:\mathbb{B}^n\mapsto\mathbb{B}$ is* negacyclic *(with respect to some variable ordering) if, when the truth table is ordered by a* most significant bit *(MSB) variable x_α, the cofactors satisfy*

$$f|_{x_\alpha=1}=\neg\big(f|_{x_\alpha=0}\big).$$

Theorem 4.3.1. *A function $f:\mathbb{B}^n\mapsto\mathbb{B}$ is negacyclic iff there exist an index $\alpha\in\{0,\ldots,n-1\}$ and*

[3] In the binary message-space setting we study here, logical negation $\neg$ and additive negation "$-$" on the torus coincide at the level of decoded bits, so we use them interchangeably.

$\varphi = \sum_{i=1}^{3}(2^{i-1}\cdot b_i)$	$\{b_1, b_2, b_3\}$	$F(w)$
0	$\{0,0,0\}$	$f(0,0,0)$
1	$\{1,0,0\}$	$f(1,0,0)$
2	$\{0,1,0\}$	$f(0,1,0)$
3	$\{1,1,0\}$	$f(1,1,0)$
4	$\{0,0,1\}$	$f(0,0,1) = \neg f(0,0,0)$
5	$\{1,0,1\}$	$f(1,0,1) = \neg f(1,0,0)$
6	$\{0,1,1\}$	$f(0,1,1) = \neg f(0,1,0)$
7	$\{1,1,1\}$	$f(1,1,1) = \neg f(1,1,0)$

Figure 4.3: Compression for three-input negacyclic gates under the radix-2 projection (Eq. (4.3)) with the disjoint input assigned as MSB. The first half of the table (shaded) is freely chosen; the second half is fixed by negacyclicity, $F(w+2^{n-1}) = -F(w)$ (logical $\neg$), so only 2^{n-1} entries need to be encoded.

a function $g : \mathbb{B}^{n-1} \mapsto \mathbb{B}$ *independent of* x_α *such that*

$$f(x) = x_\alpha \oplus g(x_{\setminus\alpha}).$$

Proof. ($\Rightarrow$) Let x_α be the MSB witnessing negacyclicity; by Shannon expansion,

$$f = (x_\alpha \wedge f_1) \vee (\neg x_\alpha \wedge f_0),$$

with $f_1 = f|_{x_\alpha=1}$ and $f_0 = f|_{x_\alpha=0}$. Since $f_1 = \neg f_0$ and f_0 is independent of x_α, we get

$$f = (x_\alpha \wedge \neg f_0) \vee (\neg x_\alpha \wedge f_0) = x_\alpha \oplus f_0.$$

($\Leftarrow$) If $f = x_\alpha \oplus g$ with g independent of x_α, then $f|_{x_\alpha=1} = \neg g = \neg\big(f|_{x_\alpha=0}\big)$, i.e., the top half of the table negates the bottom half when ordered by x_α as the MSB. □

Projection choice. For negacyclic gates we reuse the radix-2 assignment from Eq. (4.3),

$$\phi_{\text{neg}}(b_1, \ldots, b_n) = \sum_{i=1}^{n} 2^{i-1} \cdot b_i,$$

with the *disjoint* variable placed as the MSB, ensuring the truth table is negacyclic under this indexing.

Theorem 4.3.2. *Negating any subset of inputs and/or the output of a negacyclic function preserves negacyclicity.*

Proof. If $f = x_\alpha \oplus g$ with g independent of x_α, then for any subset S of variables, define ν_S to flip those bits. The composition $f \circ \nu_S$ retains the cofactor relation

$$(f \circ \nu_S)|_{x_\alpha=1} = \neg\big((f \circ \nu_S)|_{x_\alpha=0}\big),$$

since ν_S applies identically to both cofactors, and output negation (if applied) only swaps g and $\neg g$. □

Conclusion. Under $p = 8$, all three-input negacyclic gates are admissible, with the "disjoint" input variable (the x_α in Theorem 4.3.1) assigned the largest projection weight 4 (2^{3-1}), so that the truth table splits into two negated halves. By Theorem 4.3.2, input/output negations need not be enumerated separately during matching.

Gate Set Summary

For the common setting $p = 8$, we adopt as the target TFHE library:

(i) all two-input gates;

(ii) all three-input symmetric gates (optionally with one input negated); and

(iii) all three-input negacyclic gates (input/output polarities are implicit via Theorem 4.3.2).

4.3.2 Area-Oriented Technology Mapping

We represent the subject network as an *XOR–AND–inverter graph* (XAG), in which each node is either a two-input XOR or a two-input AND, and edge attributes record optional inversion. This is a natural choice for TFHE: any two-input gate is allowed, so every two-variable Boolean function can be a single XAG node.

Mapping transforms the XAG into a TFHE circuit over the gate set from Section 4.3.1. The flow has two phases:

1. **Matching.** Enumerate feasible cuts (up to size $k = 3$) at each node; for each cut, Boolean-match its local function against the admissible gate templates (with the appropriate projection/weighting to meet the $\frac{p}{2}$-slot limit).
2. **Selection.** Choose one matched cut (one gate) per node to cover the subject graph, optimizing a gate-count objective (area).

Matching

We perform standard k-feasible cut enumeration ($k = 3$) [63]. Each cut represents a candidate implementation of its root via a single gate applied to leaves at lower topological levels. Boolean matching filters the candidates: for each cut, we check membership in the admissible families (two-input, three-input symmetric, three-input negacyclic), under a legal projection ϕ that compresses the truth table to $\leq \frac{p}{2}$ slots (cf. the sub-subsection on "programmable bootstrapping" in Section 4.2.2).

Algorithm 4.1: One selection pass in technology mapping

Input: Subject graph G; cost estimator *est_cost*.
Output: G with a representative cut selected at each node.

```
1 foreach node n ∈ G in topological order do
2 |   n.cost ← +∞; n.rep ← ⊥
3 |   foreach matched cut c rooted at n do
4 |   |   c.cost ← est_cost(G, c)
5 |   |   if c.cost < n.cost then
6 |   |   |   n.cost ← c.cost
7 |   |   |   n.rep ← c
8 return G
```

Handling polarities. To increase match coverage without unnecessary duplication:

(i) we include *three-input symmetric gates with one input negated* in the library. This captures all useful input-polarity cases for symmetric functions because:

   (a) output negation preserves symmetry; and

   (b) negating n' inputs is equivalent to negating the remaining $n - n'$ inputs and the output, so enumerating more than one negated input is redundant.

(ii) for three-input negacyclic gates and all two-input gates, we do *not* enumerate explicit input/output negations during matching: by Theorem 4.3.2 (and closure of the two-input set), these polarities are naturally accounted for later (e.g., by edge inversions).

Cost of inversion. Inversions in TFHE are free of PBS (a logical NOT corresponds to a cheap torus negation). Thus, a matched "gate+inverter" has the same PBS cost as the gate alone. Missing such matches would bias the mapping toward sub-optimal decompositions.

Selection

We follow the dynamic-programming selection of [130], which assigns each node a representative (matched) cut minimizing a global area proxy. Because exact area is path-dependent, we iterate selection with two complementary heuristics — *area flow* [121] and *exact area* [130] — to balance global sharing and local accuracy.

Heuristics. *Area flow* distributes the area of shared logic across its fanouts to capture global reuse. For a matched cut c implementing node n with leaves $\ell \in \mathrm{Leaves}(c)$,

$$\mathrm{AF}(c) = 1 + \sum_{\ell \in \mathrm{Leaves}(c)} \frac{\mathrm{AF}(\ell)}{\mathrm{fanout}(\ell)},$$

where the "1" accounts for the single gate implementing n (by construction of the library). In contrast, *exact area* computes the size of the *maximum fanout-free cone* (MFFC) of n if c is selected (Chapter 2), giving a precise local contribution. We alternate passes with these estimators until the representative cuts stabilize.

Netlist extraction. After selection converges, we traverse the subject graph in reverse topological order and replace each node by its representative cut. Each selected cut instantiates one gate from the homomorphic gate library (with edge inversions as needed). The resulting netlist is a TFHE circuit with minimized gate count under the admissible gate set, ready for PBS-based evaluation.

4.4 MV-PBS-aware Mapping

This section details how we maximize the use of *multi-value programmable bootstrapping* (MV-PBS) [43]. Our support comes from two directions:

1. we integrate the "near-constant PBS cost per input set" property directly into technology mapping so that *multi-output* gates sharing the same inputs (and weight assignment) can be instantiated as a single PBS (Section 4.4.1); and
2. we perform a post-mapping *inverter reduction* to further increase opportunities to group outputs into a single multi-value PBS (Section 4.4.2).

4.4.1 Technology Mapping with Label Monitoring

MV-PBS amortizes the cost of blind rotation across several outputs, provided those outputs:

(i) depend on the same set of inputs,

(ii) use the *same projection/weight assignment* to compute the PBS index, and

(iii) belong to a gate class compatible with a common test polynomial factorization (e.g., same symmetric-family instance or same negacyclic structure; cf. Section 4.3.1).

To exploit this *during* mapping (not only as a post-process), we attach a compact *label* to every matched cut and then monitor labels as we select representative cuts. Intuitively, identical labels indicate that the cuts can be realized by a *single* MV-PBS.

Labels

During matching (cf. the sub-subsection on "matching" in Section 4.3.2), every cut that successfully matches our TFHE gate library receives a *label* summarizing the information

needed to decide MV-PBS compatibility later. For three-feasible cuts (at most three leaves), we use:

Label structure.

(i) *Ordered leaves:* three entries storing the leaf nodes in a canonical order with zero padding if the cut has fewer than three leaves.

- For **symmetric** 3-input matches, we list leaves in topological order.
- For **negacyclic** 3-input matches (Theorem 4.3.1), the *disjoint* variable must be the MSB of the weighting. We therefore put the disjoint leaf *first* and order the remaining leaves topologically. This guarantees that two labels coincide only if they choose the *same* disjoint input for the MSB position.

(ii) *Gate class/type:* one entry recording the gate family:

- `SYM` (three-input symmetric),
- `SYM_1NEG` (three-input symmetric with exactly one input negated; we also mark *which* leaf is negated. This is essential because negating an input flips the weight's sign in the projection index. Distinguishing these cases prevents false positives when grouping),
- `NEGACYCLIC` (three-input negacyclic), or
- `BIN2` (all two-input gates; we keep `BIN2` to allow MV-PBS across two-input functions with the *same* two leaves, as all two-input gates share the radix-2 weight assignment)

Why these fields suffice.

- The ordered leaves encode the *support set* and the *weight layout* (MSB position) induced by the gate class.
- The class/type disambiguates different factorization templates for the test polynomials (symmetric vs. negacyclic) and ensures we only group functions for which a common PBS blind rotation is valid.
- The polarity mark guarantees that symmetric-with-one-negated-input instances are only grouped when their projection signs match.

Two cuts can be realized as a *single* MV-PBS iff their labels are *identical*. This check is constant-time per candidate during selection.

Enhanced Exact Area Heuristic

We now fold label-awareness into selection (cf. the sub-subsection on "selection" in Section 4.3.2). The classical *exact area* computes the size of the MFFC-like contribution of a candidate cut by recursively "claiming" the nodes it newly makes necessary. We enhance this by maintaining a global `label_count` map over already-selected representative cuts, so that whenever a new candidate's label is already present, we charge *no* additional gate for the root (it will be merged into an existing gate to form a multi-output one). Only the incremental logic in its transitive fan-in cone is charged.

Algorithm 4.2 shows the procedure. The key differences from the standard exact area are:

(i) the root gate cost defaults to 1 but becomes 0 if `label_count[label]` > 0; and

(ii) we use per-node reference counters (`ref`) to avoid double-counting shared fan-in logic within the same tentative evaluation.

Discussion and correctness.

- The root's unit cost accounts for one Boolean gate, whose homomorphic evaluation invokes one PBS operation; setting it to 0 when the label is already present models "merge into an existing multi-output gate" (same inputs, same weighting, same gate class), which does *not* trigger an extra blind rotation.
- The recursion through leaves only charges logic whose representative cuts have `ref`= 0 at the time of evaluation — precisely the MFFC of the candidate *under* the current partial selection — ensuring no double counting.
- We *do not* increment `label_count` during tentative evaluations; it is updated only once when the best representative is committed. This ensures fairness across candidates.

4.4.2 Inverter Reduction

As noted in Section 4.3.2, we allow *one* input negation for three-input symmetric matches to capture more opportunities. While this improves compactness, it sometimes *prevents* grouping into a multi-output Boolean gate when two gates with the same support produce outputs that differ only by a final inversion on a shared internal signal. In such cases, pushing/pulling inversions across adjacent gates can align labels and unlock grouping.

Example 4.4.1. *In Figure 4.4 a, c_1 computes a minority; its output is inverted before feeding c_2 (to produce the next carry). That inverter blocks merging the PBS for s_2 and c_2: their labels differ due to the polarity on the shared input. By* negating *the gate functions of c_1 and s_2 (respecting Boolean identities — XOR/XNOR and majority/minority relationships), we can* absorb *the*

Algorithm 4.2: Representative cut selection guided by the enhanced exact-area heuristic

Input: Node n, subject graph G, global label counter *label_count*
Output: Node n with representative cut set; counters updated

```
1  n.cost ← +∞
2  foreach matched cut c rooted at n do
3  |  c.cost ← ref_cut(G, c)
4  |  if c.cost < n.cost then
5  |  |  n.cost ← c.cost; n.rep ← c
6  |  deref_cut(G, c)
   // Commit the best candidate and update global counters
7  ref_cut(G, n.rep); n.ref ← 1
8  label_count[n.rep.label] ← label_count[n.rep.label] + 1
9  Function ref_cut(G, c) :
10 |  cost ← 1
11 |  if label_count[c.label] > 0 then
12 |  |  cost ← 0
13 |  foreach leaf l of c do
14 |  |  if l is a PI then
15 |  |  |  continue
16 |  |  l.ref ← l.ref + 1
17 |  |  if l.ref = 1 then
18 |  |  |  cost ← cost + ref_cut(G, l.rep)
19 |  return cost
20 Function deref_cut(G, c) :
21 |  foreach leaf l of c do
22 |  |  if l is a PI then
23 |  |  |  continue
24 |  |  l.ref ← l.ref − 1
25 |  |  if l.ref = 0 then
26 |  |  |  deref_cut(G, l.rep)
```

inverter and obtain Figure 4.4 b, where s_2 and c_2 now share the same label. In the 128*-bit adder, this flip is available across almost all adjacent sum/carry pairs, increasing the number of pairs merged by MV-PBS and reducing the total PBS count from* 191 *to* 128 *in this instance.*

Post-mapping procedure. Starting from the mapped network, traverse nodes in topological order and try to absorb input inversions of three-input symmetric gates back into their driving gates:

1. Let n be a 3-input symmetric gate with exactly one negated input, driven by node l. Because output inversion does not change the gate class (cf. the sub-subsection on "matching" in Section 4.3.2), we may push the inversion into l by negating l's gate

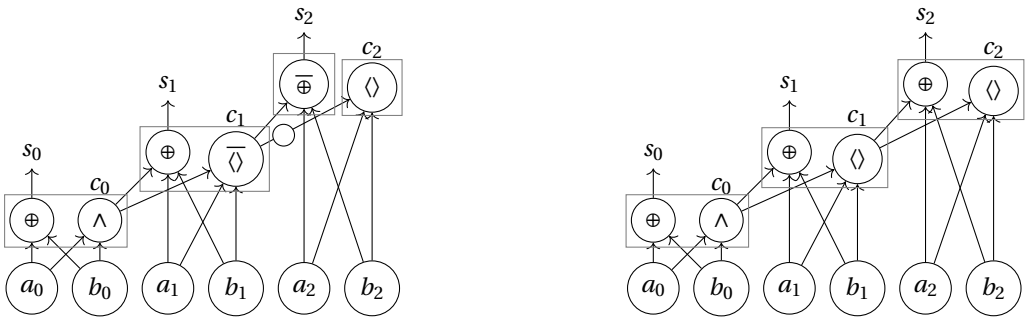


Figure 4.4: Least significant three bits of a mapped 128-bit adder, (a) *before* and (b) *after* post-mapping inverter reduction. Gates boxed together share the same support and weighting and can be grouped into a single multi-output gate (MV-PBS). Bubbles on edges indicate logic complementation. '⟨⟩' and '$\overline{⟨⟩}$' indicate Boolean majority and minority operations, respectively.

function — provided doing so does not introduce extra inverters elsewhere.

2. If l has *one* fanout (namely n), commit the absorption (safe: no other sinks are affected).

3. If l has *multiple* fanouts, we tentatively flip l and check the other fanouts $n' \neq n$:

   (i) If n' is a **two-input** gate or a **three-input negacyclic** gate, input polarity changes *do not* alter gate family/type (Theorem 4.3.2); the new inversion can be absorbed at n' by adjusting its (free) output polarity, hence no new inverter remains.

   (ii) If n' is **three-input symmetric**: *skip* the absorption only when

   (a) the edge $l \to n'$ is already negated (two inversions cancel), or

   (b) n' is XOR/XNOR (the polarity can be absorbed by toggling XOR↔XNOR without inserting a separate inverter).

   In all other symmetric cases, flipping l would force us to add an inverter on $l \to n'$, so we do *not* commit.

This targeted rewrite aligns input polarities across sibling cones, increasing the chance that downstream cuts acquire *identical labels* and hence become groupable by MV-PBS in the final netlist.

Why post-mapping (not during matching)? The configuration in Figure 4.4 a often arises because the *subject graph* presents certain nodes in complemented form; allowing input negations during matching is crucial for area optimality (no PBS cost for inverters). The observed misalignment is thus an inevitable by-product of optimal mapping. A dedicated post-pass that globally examines fanout interactions is the right place to reconcile polarities without harming area.

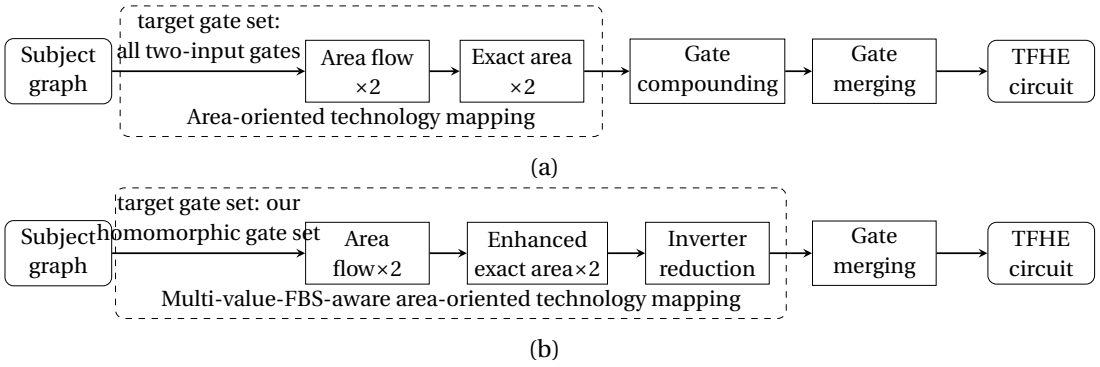


Figure 4.5: Homomorphic Boolean circuit synthesis flows evaluated in this work: (a) state-of-the-art (SOTA) and (b) ours. Both end with a merging pass that combines qualified single-output gates into multi-output gates to unlock MV-PBS.

Summary. Labels make MV-PBS a first-class optimization objective *within* mapping (via enhanced exact area), while inverter reduction cleans up polarity mismatches *after* mapping. Together, they systematically increase the number of outputs merged per PBS, directly reducing blind rotations and end-to-end homomorphic evaluation time.

4.5 Experimental Evaluations

This section evaluates the proposed homomorphic Boolean circuit synthesis approach in two stages. First (Section 4.5.1), we profile circuit-level outcomes: the size of the synthesized TFHE circuits after merging (which directly correlates with the number of PBS operations), the merge rate (how effectively single-output gates are combined into multi-output gates to exploit MV-PBS), and synthesis runtime. Second (Section 4.5.2), we estimate the overall impact on homomorphic circuit evaluation efficiency by deriving per-PBS costs with a TFHE compiler and aggregating them over the synthesized circuits. Unless otherwise stated, all experiments were conducted on an Apple M1 Max with 32GB RAM.

4.5.1 Profiling Synthesized Circuits

We compare the following three flows that differ in how they *introduce larger fan-in gates* and in whether the mapping step is *MV-PBS aware*:

1. **SOTA.** Because the reference tool `AutoHoG` is not open-source, we re-implemented it based on [90]; the flow is shown in Figure 4.5 a. First, area-oriented technology mapping targets *only two-input* gates. Next, larger fan-in gates are introduced *locally* by *compounding* pairs of adjacent two-input gates into a three-input gate when a local structure allows it. Finally, a *gate-merging* pass combines single-output gates that share

   (i) identical supports, and

(ii) the same gate type and weight assignment

into multi-output gates, enabling *MV-PBS* (cf. the sub-subsection on "multi-value programmable bootstrapping" in Section 4.2.3).

2. **TechMap.** To isolate the benefit of leveraging larger fan-in gates *via technology mapping* (rather than via local compounding), we modify SOTA so that mapping targets the homomorphic gate set from Section 4.3.1 *directly* (all two-input gates, all three-input symmetric gates, all three-input negacyclic gates). The downstream compounding stage and the merging stage are retained, so that MV-PBS could be exploited.

3. **Enhanced TechMap.** Our full flow (Figure 4.5 b). As in TechMap, the mapping targets the proposed homomorphic gate set. In addition, the *selection* step adopts the *enhanced exact area* heuristic with *label monitoring* (Section 4.4.1), making mapping *explicitly aware* of MV-PBS opportunities. Finally, we perform a post-mapping *inverter-reduction* pass (Section 4.4.2), which both reduces spurious inverters and can expose additional merging opportunities.

Benchmarks and infrastructure. We use the EPFL combinational benchmark suite [5], which includes ten arithmetic and ten random/control designs. Subject graphs are represented as XAGs: instead of adopting the MC-optimized version (cf. Section 2.3.1) as the baseline — as MC is irrelevant to the cost model of this chapter — we convert the original *AND-inverter graphs* (AIGs) to XAGs with the mapper of [169]. All flows (and the pre-processing) are implemented in C++ within `mockturtle` [165].

Table 4.1 reports the number of gates after merging ("#Gates"), the merge rate ("Merge"), and the synthesis runtime ("Time"). We define the merge rate as

$$\text{Merge} = \frac{\#\text{gates before merging} - \#\text{gates after merging}}{\#\text{gates before merging}},$$

so, a higher value indicates that more single-output gates were folded into multi-output gates, and thus that more MV-PBS opportunities were realized. Since each gate triggers one PBS, #Gates is a first-order proxy for homomorphic circuit evaluation effort.

TechMap vs. SOTA. The key structural difference is how three-input gates are introduced. SOTA uses *local compounding* of adjacent two-input gates; TechMap *maps directly* to the larger gate set (Section 4.3.1). Across all benchmarks, TechMap produces circuits with post-merge gate counts that are never worse and usually strictly better than SOTA. This indicates that greedy, local compounding lacks a global perspective during cut selection and can prematurely lock in sub-optimal choices. Although omitted from Table 4.1 for space, we observe that SOTA instantiates on average 1.92× *more* three-input gates than TechMap yet still yields *larger* post-merge designs — evidence that "more large gates" do not necessarily lead to "better design"

Table 4.1: Evaluating the three homomorphic circuit synthesis flows. *#Gates* is the post-merge gate count (i.e., number of PBS operations). *Merge* is the merge rate; *Time* is the synthesis runtime. Lower #Gates and Time are better; higher Merge is better.

Benchmark	*SOTA*			*TechMap*			*Enhanced TechMap*		
	#Gates	Merge[%]	Time[s]	#Gates	Merge[%]	Time[s]	#Gates	Merge[%]	Time[s]
adder	507	20.16	<0.01	191	25.39	<0.01	128	50.00	<0.01
barrel shifter	2 496	7.45	0.01	2 496	7.45	0.01	2 496	7.45	0.01
divider	28 773	0.10	0.08	13 116	0.31	0.06	13 076	0.82	0.06
hypotenus	121 325	3.72	0.55	83 194	8.82	0.36	78 076	14.56	0.40
log2	20 160	2.09	0.05	14 058	6.34	0.05	13 573	9.62	0.06
max	2 154	5.77	0.01	2 068	8.70	0.01	2 066	11.41	0.01
multiplier	14 530	3.74	0.04	10 442	8.60	0.03	9 957	12.85	0.04
sine	3 578	1.65	0.01	2 502	6.64	0.01	2 398	10.92	0.01
square-root	10 452	1.19	0.03	8 276	17.37	0.03	8 218	17.95	0.03
square	11 968	2.09	0.03	8 661	7.12	0.02	7 547	19.11	0.03
round-robin arbiter	11 434	0.00	0.06	11 605	0.00	0.07	11 605	0.00	0.08
coding-cavlc	554	9.62	<0.01	545	8.86	<0.01	542	11.15	<0.01
ALU control unit	97	11.82	<0.01	94	8.74	<0.01	94	10.48	<0.01
decoder	291	3.96	<0.01	292	2.34	<0.01	292	3.95	<0.01
i2c controller	1 109	2.89	<0.01	1 032	2.46	<0.01	1 029	3.38	<0.01
int to float converter	175	8.85	<0.01	171	8.56	<0.01	170	11.46	<0.01
memory controller	36 446	2.70	0.28	35 510	2.96	0.12	35 016	5.45	0.13
priority encoder	833	0.12	<0.01	818	0.12	<0.01	818	0.24	<0.01
look-ahead XY router	174	1.14	<0.01	134	4.96	<0.01	126	11.27	<0.01
voter	5 170	12.55	0.03	3 306	23.98	0.01	2 936	33.15	0.01
Average	13 611.30	5.08	0.059	9 925.55	7.99	0.039	9 508.15	12.26	0.435
Norm.	**1**	**1**	**1**	**0.729**	**1.572**	**0.661**	**0.699**	**2.413**	**0.737**

if without a global selection objective. Overall, TechMap reduces post-merge gate count by 27.08% on average and shortens synthesis time by 33.90%. This is because, in TechMap, all decisions are made once during cut selection on the subject graph (cf. the sub-subsection on "selection" in Section 4.3.2) and the TFHE circuit is materialized in a single pass.

Enhanced TechMap vs. TechMap. Adding label monitoring to exact area introduces only a modest 1.15× runtime overhead, and the overall flow still outperforms SOTA in runtime by 26.27% on average. Most importantly, the *enhanced exact area* heuristic *consistently* raises the merge rate and yields *more compact* post-merge circuits: we observe an additional 4.20% gate-count reduction over TechMap, for a total of 30.15% relative to SOTA. Interestingly, before merging (numbers omitted for space), Enhanced TechMap's pre-merge gate count is, on average, slightly *higher* ($\approx 0.36\%$) than TechMap's. This shows that optimizing solely for pre-merge gate count can be misleading when MV-PBS is available: making mapping *MV-PBS aware* at selection time is what delivers superior post-merge designs.

4.5.2 Estimating Evaluation Cost Reduction

The design produced by Enhanced TechMap (Figure 4.4 b) comprises one half adder at the LSB (one two-input AND and one two-input XOR) and 127 full adders (each with a three-input majority and a three-input XOR). All these gates are symmetric; using unit weights yields $p = 6$

for the half adder and $p = 8$ for the full adder. Moreover, the compressed four-entry tables of MAJ and XOR are $^{\#}1100$ and $^{\#}1010$. Because the leftmost bit is the negation of the rightmost bit under the radix-2 layout, these tables are *negacyclic*, letting us reduce to $p = 6$. This relaxation lowers the per-PBS cost relative to designs that require $p = 8$. Such effects are invisible to a pure gate-count metric and motivate a more refined cost estimate.

Methodology. We estimate per-PBS cost using the TFHE compiler `Concrete` [198]. Given a synthesized circuit, we:

(i) determine the projection (weights) and the Euclidean norm of each PBS index (including the scaling applied for MV-PBS [43]),

(ii) let `Concrete` select parameters that allocate sufficient noise budget for correctness at the target security level, and

(iii) obtain a per-PBS cost estimate under those parameters.

The circuit-level estimate is then

(per-PBS cost) × #PBS (i.e., post-merge gate count).

Results. The "Eval." column in Table 4.2 shows that *Enhanced TechMap* reduces homomorphic evaluation cost by 29.94% on average compared to SOTA, with a best-case reduction of 76.02% on `adder`. These results confirm that

(i) mapping directly to an expressive homomorphic gate set, and

(ii) making selection *MV-PBS aware*

together deliver substantially faster homomorphic circuit evaluation — not merely smaller gate counts.

4.6 Discussion

This section distills the main takeaways from our study and sketches several directions for future work. We first reflect on the strategic choice of an *incomplete* yet principled homomorphic gate set for TFHE circuit synthesis, and then discuss implications for a broader class of logic synthesis problems that arise uniquely in TFHE due to PBS and the cost structure of multi-output gates.

Table 4.2: Estimated evaluation cost of synthesized TFHE circuits. "Eval. [ms]" denotes the estimated time required to homomorphically evaluate the TFHE circuit; lower is better.

Benchmark	*SOTA*	*TechMap*	*Enhanced TechMap*
	Eval. [ms]	Eval. [ms]	Eval. [ms]
`adder`	20 280	7 258	4 864
`barrel shifter`	99 840	99 840	99 840
`divider`	1 150 920	524 640	523 040
`hypotenus`	4 853 000	3 327 760	3 123 040
`log2`	806 400	562 320	542 920
`max`	86 160	82 720	82 640
`multiplier`	581 200	417 680	398 280
`sine`	157 432	110 088	105 512
`square-root`	418 080	331 040	328 720
`square`	478 720	346 440	301 880
`round-robin arbiter`	457 360	464 200	464 200
`coding-cavlc`	22 160	21 800	21 680
`ALU control unit`	3 880	3 760	3 760
`decoder`	11 640	11 680	11 680
`i2c controller`	44 360	41 280	41 160
`int to float converter`	7 000	6 840	6 800
`memory controller`	1 603 624	1 562 440	1 540 704
`priority encoder`	33 320	32 720	32 720
`look-ahead XY router`	6 960	5 360	5 040
`voter`	227 480	145 464	117 440
Average	553 490.80	405 266.50	387 796.00
Norm.	**1**	**0.732**	**0.701**

4.6.1 Strategic Use of Incomplete Gate Sets

We augmented the `TFHE` library's default gate set with two families of three-input gates that admit strong truth-table compression under PBS: *symmetric* gates and *negacyclic* gates (Section 4.3.1). For the common configuration with plaintext space $\mathbb{Z}_8$, this tailored set is functionally complete for our technology-mapping flow. When the plaintext space grows, however, the universe of valid gates expands combinatorially [90], making a full, static gate catalogue impractical.

Our experimental evidence (Section 4.5.1) indicates that *maximizing* the count of large-fan-in gates is not the right goal; rather, the goal is to *use larger-fan-in gates strategically* to *minimize the mapped gate count* (and therefore the number of PBS operations). Because local compounding is greedy by design, `AutoHoG` tends to instantiate more three-input gates than our mapper, yet still produces *less compact* circuits on average. In contrast, our gate-set design — grounded in symmetry and negacyclicity — enables global, cut-based selection during technology mapping, so the expressiveness of larger-fan-in gates is applied exactly where it pays off in area (namely, gate count).

The broader lesson is that, for TFHE circuit synthesis, a *well-chosen but incomplete* gate set can be more effective than an exhaustive one: it keeps matching tractable, supports global

optimization (Section 4.3.2), and avoids the "greedy local optimum" pitfall of compounding. Going forward, one could enrich the gate set with additional classes that also compress well under PBS — e.g., *threshold* [137] or *near-symmetric* functions — without sacrificing tractability. Another promising angle is *adaptive gate-set selection*: profiling the subject graph to enable just those higher-arity gates that appear frequently (or promise high mergeability), while retaining a small and optimization-friendly gate library.

4.6.2 Exploring a Unique Logic Synthesis Problem

As noted in Section 4.2.4, TFHE circuit synthesis diverges from classical hardware logic synthesis because multi-output gates can be evaluated by (nearly) the *same-cost* PBS as single-output gates (cf. the sub-subsection on "multi-value programmable bootstrapping" in Section 4.2.3). This collapses a traditional area–fanout trade-off and yields a distinct optimization landscape: we want to *maximize mergeability* (shared supports, identical gate types) subject to global area optimality.

Our mapping flow embraces this through *label-aware* selection. Concretely, we left the global *area flow* heuristic intact to retain its role as a fast, coarse global guide (cf. the sub-subsection on "selection" in Section 4.3.2), but we *enhanced the exact-area phase* (Section 4.4.1) with label monitoring that detects when newly chosen cuts can be realized jointly as MV-PBS, charging zero marginal area in those cases. Empirically, attempting to "pre-load" merge-awareness into the *early* area flow phase did not improve results: at that stage, too few representative cuts are fixed to make reliable merge predictions. By contrast, exact area runs *after* area flow, when many choices are locally decided and small changes can deterministically unlock merges.

These observations suggest several concrete research directions:

- **Merge-aware global guidance.** Design a softened, probabilistic mergeability signal for area flow (e.g., via learned or analytical estimates of label co-occurrence) so that global guidance remains stable yet more "merge-attentive."
- **Cost models that reward merge potential.** Extend the objective beyond pure gate count to a multi-objective score that balances area against an explicit mergeability term (e.g., expected reduction in PBS invocations), while preserving the tractability of dynamic programming.
- **Adaptive weighting and label design.** Co-optimize the projection weights (for symmetric/negacyclic compression) with mapping choices to increase label matches across cones, thereby creating more MV-PBS opportunities.
- **Integration with post-mapping cleanup.** Our inverter-reduction pass (Section 4.4.2) increases merge opportunities without changing gate count. A tighter loop — alternating localized re-selection with inverter normalization — may expose additional merges that a single post-pass cannot.

- **Beyond $\mathbb{Z}_8$.** For larger plaintext spaces, develop taxonomy-driven gate curation (e.g., classes defined by symmetry groups or negacyclic constraints on $\mathbb{Z}_p$) plus scalable matching that preserves the benefits we observed at $\mathbb{Z}_8$.

Summary. In summary, TFHE mapping with tailored gate-set and MV-PBS awareness is its own *logic synthesis problem*, not merely a minor variant of hardware mapping. The combination of cut-based technology mapping, label-aware exact-area refinement, and lightweight post-mapping normalization forms a practical baseline. We expect that merge-aware global heuristics and adaptive gate-set design will push the frontier further, enabling still more efficient TFHE-based homomorphic evaluation of Boolean functions.

5 Encoding Strategy Management for TFHE: An ESOP-Guided Approach

5.1 Motivation

A different knob: switching plaintext spaces. Chapter 4 focused on TFHE circuit synthesis when the *plaintext space is fixed* for the entire circuit, improving evaluation by curating a TFHE-amenable gate set (symmetric and negacyclic gates) and exploiting *multi-value programmable bootstrapping* (MV-PBS). In this chapter, we study a different design knob: the plaintext space used by a *programmable bootstrapping* (PBS) *need not match* that of its input ciphertext. A PBS can evaluate a small function (via a *look-up table* (LUT)) *and* re-encode the result into a different plaintext space. This enables *dual-space* (more generally, multi-encoding) evaluation: keep signals in the Boolean space $\mathbb{B}$ to leverage essentially *free XOR*, and switch to a larger $\mathbb{Z}_p$ *only* where a PBS operation can also collapse non-linear structure effectively.

Why switching helps—and why it is nontrivial. TFHE exposes a tension:

- **In $\mathbb{B}$:** XOR is cheap (TLWE additions), but non-linear gates (typically, Boolean AND gates) have no direct LUT encoding in $\mathbb{B}$; they require a PBS into a larger space. We refer to such a PBS as a *switch PBS*.
- **In $\mathbb{Z}_p$:** A single PBS can realize the homomorphic evaluation of a non-linear gate while refreshing noise, but its cost grows with p and with the norm of the linear index used to query the LUT. Specifically, we distinguish such PBS operations as *compute PBS* operations.

Therefore, it is attractive to remain in $\mathbb{B}$ wherever possible, and to switch into $\mathbb{Z}_p$ *only* at non-linear regions where the reduction in PBS count outweighs the higher per-PBS cost. The difficulty is determining *when and where* to switch: every switch PBS — whose purpose is only to change encoding — costs roughly as much as a compute PBS, so poorly placed switches can erase the benefit.

Why a new synthesis view is needed. A fixed-encoding flow (as in the previous chapter) cannot exploit this knob, while ad-hoc switching policies (e.g., "switch for each AND and switch back after") over- or under-convert:

- Non-linear nodes with mixed fanout (feeding both XOR and AND) induce conflicting space requirements downstream.
- Signals that feed multiple non-linear cones (represented by AND trees) of different arities risk repeated re-encoding unless conversions are coordinated globally.
- Splitting a large-arity AND tree across multiple PBS operations increases PBS count and can introduce extra switch PBS at the splits; keeping the tree intact for a single PBS may demand a larger plaintext space p.

These patterns show that encoding strategy is tightly coupled to logic structure; we need a synthesis method that reasons about *structure and encoding cost jointly*, instead of bolting switching onto a pre-existing gate netlist.

ESOP as the structural scaffold for dual-space evaluation. We adopt the *exclusive-or sum of products* (ESOP) form as our working abstraction. ESOP separates a Boolean function into product terms (cubes — ANDs of literals) and a single XOR aggregation:

- Each cube can be implemented by *one compute PBS* in a suitably chosen $\mathbb{Z}_p$ (with p guided by cube arity and the index norm).
- The XOR of cube outputs is then performed *in* $\mathbb{B}$ by free-XOR after the encoding switches back to $\mathbb{B}$.

This representation exposes clean boundaries at which to place encoding transitions, and it provides levers to trade off:

(i) the number of cubes (PBS count)

(ii) cube arities (which influence p and per-PBS cost), and

(iii) the set of places where switches are actually needed.

Cost model and synthesis objective. We formalize a cost model that prices *both* compute PBS and switch PBS. It incorporates gate type, chosen p, and the Euclidean norm of the linear index. Given this model, our goal is to *partition* a function into ESOP-aligned regions, *select* cube structures and encoding spaces, and *place* the minimal number of switches so that total cost is minimized.

Scope relative to the previous chapter. This chapter is an *alternative* to the single-space mapping of Chapter 4, not a composable extension. Once we allow encoding transitions, the non-linear parts that we choose to evaluate in a larger plaintext space are compiled *directly* to compute PBS operations; there is no opportunity (and no obvious way) to resynthesize these regions using the TFHE-native gate set from the previous chapter. Consequently:

- We *do not* reuse the gate-level technology mapping of Chapter 4 within large-space regions.
- Our optimization acts at the ESOP and encoding-schedule level (cube shaping, p selection, and switch placement), with the XOR layer handled in $\mathbb{B}$.

At the thesis level, the two chapters address *distinct operating modes* for TFHE: Chapter 4 targets *single-space* designs with rich gate mapping; this chapter targets *dual-space* designs where encoding switches are first-class decisions. They are complementary in scope but *not intended to be combined into a single synthesis pipeline.*

Empirical outlook. As shown later in Section 5.5, this encoding-aware, ESOP-guided approach delivers substantial speedups over baselines that either fix the encoding throughout or reduce multiplicative complexity without explicit management of encoding transitions — up to 53.46% and an average of 23.34% reduction in homomorphic evaluation time on general-purpose Boolean benchmarks.

The technical content of this chapter is based on our prior work presented in [191].

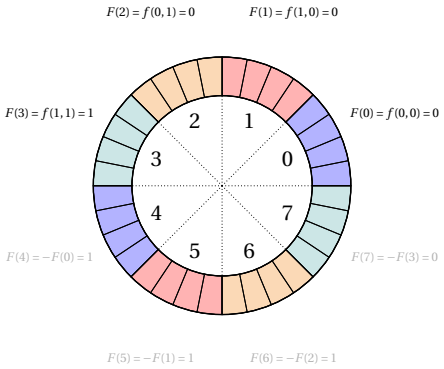
5.2 Preliminaries

This section reviews concepts needed for encoding-switch-aware TFHE circuit synthesis. We first recall how TFHE evaluates Boolean circuits via *programmable bootstrapping* (PBS) operations and summarize prior work on gate-set–driven compression and MV-PBS (Section 5.2.1). We then formalize multi-encoding-space evaluation and why managing encoding transitions is a synthesis problem in its own right (Section 5.2.2).

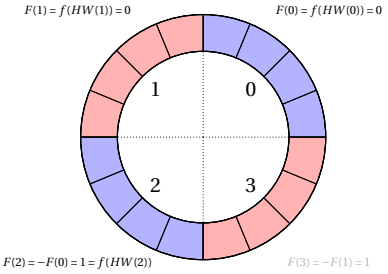
5.2.1 TFHE Circuit Synthesis: Fundamentals and Prior Work

TFHE circuits and programmable bootstrapping. A *TFHE circuit* is a Boolean circuit that serves as the blueprint for homomorphically evaluating a Boolean function under TFHE. In TFHE, each logic gate is implemented by a PBS operation, which both refreshes noise and evaluates a small function encoded as a *look-up table* (LUT).

We give an informal view sufficient for synthesis; all quantities manipulated during evaluation are encrypted *torus LWE* (TLWE) ciphertexts, but for exposition we describe computations as



(a) Naïve encoding ($p = 8$, $N = 16$). Reproduced from Figure 4.1 for convenience.



(b) Compressed encoding ($p = 4$, $N = 8$).

Figure 5.1: Encoding a two-input AND LUT on the polynomial ring $\mathbb{T}[X]/(X^N+1)$. Dotted arcs mark plaintext-space boundaries on the ring. Colored segments indicate distinct LUT entries; each freely assigned entry and its negacyclically determined partner share a color. Entries that are never indexed by design are faded.

if on plaintext. For rigorous cryptographic details, see [57] and optimizations such as [21, 22].

Conceptually, evaluating an n-input Boolean gate $f : \mathbb{B}^n \mapsto \mathbb{B}$ by PBS proceeds in two logical stages:

1. compute an *index* m that encodes the current input pattern via a *projection function* $\phi : \mathbb{B}^n \to \mathbb{Z}_p$; and

2. query the LUT $F : \mathbb{Z}_p \to \mathbb{B}$ so that

$$f(b_1, \ldots, b_n) = F\big(\phi(b_1, \ldots, b_n)\big)$$

for all $(b_1, \ldots, b_n) \in \mathbb{B}^n$. Correctness then reduces to ensuring the encrypted LUT is indexed by the encrypted m produced from the encrypted inputs.

The LUT is embedded in a degree-N polynomial ring $\mathbb{T}[X]/(X^N+1)$, where $\mathbb{T}$ is the torus. Because $X^N = -1$, the ring is *negacyclic*: among the $2N$ coefficient "slots," only N can be assigned freely; the remaining N are fixed by negacyclic symmetry. Formally, for any index $m > \frac{p}{2} - 1$ we must have

$$F(m) = -F\left(m - \tfrac{p}{2}\right),$$

so, each LUT entry is mirrored with a sign flip in the second half. In practice, each value $F(m)$ is replicated across $\frac{2N}{p}$ consecutive coefficients to equalize noise. The encrypted index m drives a *blind rotation* that positions the accumulator under the appropriate replicated block, thereby returning the desired (encrypted) LUT value.

Without compression, the LUT has 2^n entries and a standard choice for ϕ is the radix-2 weighted sum,

$$\phi(b_1, \ldots, b_n) = \sum_{i=1}^{n} 2^{i-1} b_i,$$

which gives a unique index per input pattern. Figure 5.1 a shows this naïve encoding for a two-input AND with $\phi(b_1, b_2) = b_1 + 2b_2$ into $\mathbb{Z}_8$; with $N = 16$, each LUT entry is backed by four coefficients. Because the maximum index is 3, all accessed indices lie in the first half of the ring; the second half comprises "don't-care" values that do not affect correctness.

Gate-set–driven LUT compression. Two techniques underpin compact TFHE circuits. The first exploits *LUT compression* by choosing ϕ and gate classes whose truth tables admit many merged entries. Chapter 4 of this thesis systematizes this via Boolean properties — especially *symmetry* and *negacyclicity* — to precompute a library of functions compressible under a fixed p, along with their projections ϕ. This turns "large-fan-in exploitation" into a standard technology mapping problem and avoids runtime, greedy compounding decisions as in [90, 40].

As introduced in the sub-subsection on "symmetric gates" in Section 4.3.1, symmetric gates illustrate the benefit. For an n-input symmetric function, the output depends only on Hamming weight, so a *uniform-weight* projection

$$\phi(b_1, \ldots, b_n) = \sum_{i=1}^{n} b_i$$

maps all inputs of the same weight w to the same LUT entry $F(w)$, reducing the table to $n+1$ entries. Moreover, many symmetric gates satisfy $F(n) = -F(0)$ (and more generally, $F(w+n/2) = -F(w)$ when applicable), allowing further compression by negacyclic symmetry (cf. the sub-subsection on "negacyclic gates" in Section 4.3.1). Figure 5.1 b shows the two-input AND under this scheme: compared to the naïve encoding in Figure 5.1 a, the compressed encoding achieves the same replication per entry with smaller N (8 instead of 16) and smaller p (4 instead of 8), hence a cheaper PBS. For larger fan-in, the same principle permits implementing gates beyond two inputs in a single PBS, often reducing gate count substantially.

Generalization to larger AND trees. Following the same construction as in Figure 5.1 b, a $(n+1)$-input AND tree (realized by n two-input AND gates) can be evaluated in a *single* PBS by using the uniform-weight projection, which maps each input pattern to its Hamming weight. The LUT for an $(n+1)$-input AND has $n+2$ entries indexed by $w \in \{0, \ldots, n+1\}$, with $F(w) = 0$ for $w < n+1$ and $F(n+1) = 1$. Leveraging the negacyclic constraint $F(w + p/2) = -F(w)$,

choosing

$$p = 2(n+1)$$

suffices: the freely programmable region $\{0,\dots,p/2-1\} = \{0,\dots,n\}$ contains all zero entries, while the index $w = p/2 = n+1$ carries the single one, and the remaining n positions are fixed by negacyclic symmetry. In practice, pick a ring degree N such that $p \mid 2N$ so that each LUT value is replicated uniformly $2N/p$ times. Under this encoding, the whole $(n+1)$-input AND tree is computed with one compute PBS in $\mathbb{Z}_{2(n+1)}$, replacing the n PBS calls that would be required if each two-input AND were bootstrapped separately.

Multi-value PBS (multi-output gates). The second technique is *multi-value programmable bootstrapping* [43], which evaluates multiple functions sharing the same supports within *one* PBS. Algebraically, the test polynomials (test vectors) are factored into a common component (driving a single blind rotation) and function-specific factors recovered by light polynomial multiplications. Synthesis flows can therefore prefer *multi-output gates* (several single-output gates with identical supports and compatible encodings), amortizing the dominant blind-rotation cost. Recall that, in Chapter 4, we integrate this idea by biasing technology mapping toward patterns that enable shared-input, multi-output realizations.

5.2.2 Multi-Encoding-Space Homomorphic Evaluation

The techniques above implicitly assume that *every* logic gate is realized by a PBS. An orthogonal direction reduces PBS usage by performing as many operations as possible via efficient TLWE arithmetic (addition and scalar multiplication) without bootstrapping, i.e., *leveled* operations.

Under the Boolean plaintext space $\mathbb{B}$, homomorphic XOR is simply ciphertext addition and is extremely efficient ("free-XOR"). In contrast, realizing AND as TLWE multiplication is both costly and noise-expensive (quadratic noise growth), so a PBS must still follow each multiplication to refresh the ciphertext [57], undermining the benefit.

A natural compromise is to evaluate *linear* structure (XOR) in $\mathbb{B}$ via leveled operations and evaluate *non-linear* structure directly by PBS — but PBS LUTs for non-linear gates generally do *not* fit into $\mathbb{B}$; for instance, Figure 5.1 b shows how to evaluate a two-input AND gate, the most basic non-linear logic primitive by invoking a PBS operation in $\mathbb{Z}_4$. Recent work [28] shows that a PBS can also *switch* plaintext spaces by emitting the output in a target space encoded in the accumulator; thus, linear portions can stay in $\mathbb{B}$ while non-linear regions are evaluated in a larger $\mathbb{Z}_p$ and, if needed, switched back.

This flexibility introduces a synthesis problem we call *encoding strategy management*: when to switch, where to switch, and how to restructure logic so that the performance gains of free-XOR are not erased by excessive *switch PBS*. Poorly placed switches can easily outnumber saved compute PBS, especially near nodes with mixed fanout (feeding both XOR and AND logic) or

when the same signal would otherwise be switched repeatedly to service several non-linear cones. The central goal of this chapter is to formulate and solve TFHE circuit synthesis under multi-encoding evaluation with an explicit cost model that accounts for *both* compute PBS and switch PBS.

5.3 When to Switch: Analyzing Encoding Strategy Transitions

This section develops a systematic understanding of *encoding strategy management* — when and where switching plaintext spaces is beneficial in TFHE-based homomorphic evaluation. We focus on two circuit representations widely used in synthesis: *XOR-AND–inverter graphs* (XAGs) and the *exclusive-or sum of products* (ESOP) form. By examining how structural and functional properties interact with encoding choices, we derive practical principles that will inform the algorithmic framework in the next section.

Compute PBS vs. switch PBS. Under a multi-encoding-space setting, each PBS operation plays exactly one of two roles:

(i) a *compute PBS*, which evaluates a logic operation (e.g., an n-input gate), possibly *also* switching the output into a chosen space; or

(ii) a *switch PBS*, whose sole purpose is to convert a `LWE` ciphertext from one plaintext space to another without performing logic.

Because both types have comparable runtime cost, an effective synthesis must not only reduce the number of compute PBS operations, but also aggressively *avoid* switch PBS whenever the same effect can be achieved by restructuring the computation or unifying encoding choices.

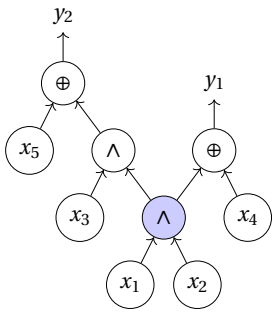
5.3.1 Encoding Strategy Behavior in XAG-Based Evaluation

In an XAG, internal nodes are two-input XOR or AND gates; negations are edge attributes (see Chapter 2 for circuit notation). Under TFHE, XOR is "free" in $\mathbb{B}$ (TLWE additions and scalar multiplications), while AND requires a PBS. Consequently, encoding transitions tend to appear around AND/XOR boundaries, especially when an AND output fans out to both XOR and AND consumers (*mixed fanout*).

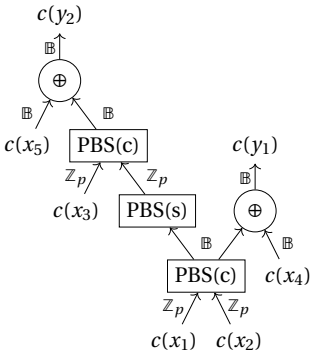
AND gates with mixed fanout

XOR-only fanout. If an AND output drives only XOR gates, that AND is the root of its AND tree. A single *compute PBS* at that node can

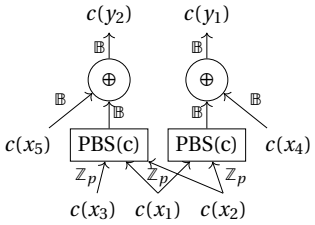
(i) evaluate the AND tree, and



(a) Representative XAG fragment. The mixed-fanout AND is highlighted in blue.



(b) Split strategy: one switch PBS + two compute PBS.



(c) Preserve strategy: two compute PBS; no switch PBS.

Figure 5.2: Handling encoding transitions for an AND gate with mixed fanout. In all panels, $c(\cdot)$ denotes a TLWE ciphertext; edges are annotated by the current plaintext space; PBS nodes are labeled as compute PBS ("PBS(c)") or switch PBS ("PBS(s)").

(ii) emit its result directly in $\mathbb{B}$, enabling downstream free-XOR without any switch PBS.

Mixed fanout. If an AND feeds both XOR and AND gates (Figure 5.2 a), two evaluation strategies arise:

(i) *Split strategy* (Figure 5.2 b): treat the lower mixed-fanout AND as the root of a *smaller* AND sub-tree; evaluate both sub-trees with separate compute PBS calls. Because the lower sub-tree result is needed both in $\mathbb{B}$ (for XOR) and in a larger $\mathbb{Z}_p$ (as input to the upper AND), a *switch PBS* is required. This yields two compute PBS and one switch PBS.

(ii) *Preserve strategy* (Fig. 5.2 c): treat the whole structure as a *single* AND tree rooted at the highest AND. This maintains a consistent encoding throughout, requires only two compute PBS, and avoids the switch PBS entirely. Although the required plaintext space grows with the tree arity (the LUT must fit the unit-weight projection over Hamming weights, with negacyclic pairing as shown in Figure 5.1 b), the small increase in each compute PBS cost is typically outweighed by removing an entire switch PBS.

Guideline. Prefer *preserving* AND trees over splitting them; the avoided switch PBS usually dominates the modest increase in the compute PBS cost.

Primary inputs with mixed fanout

A PI that drives only XOR gates may remain in $\mathbb{B}$; one that drives only AND trees may be placed directly in the unified $\mathbb{Z}_p$ used for non-linear evaluation. The challenge is *mixed-fanout PIs* that feed XOR and several AND trees (potentially of different arities): naïvely, the PI must be replicated across multiple encoding spaces, implying repeated switches.

Guideline. *Unify* the plaintext space for *all* non-linear gates to a single $\mathbb{Z}_p$ determined by the largest AND tree in the circuit. Then, each mixed-fanout PI requires at most one switch PBS (from $\mathbb{B}$ to the unified $\mathbb{Z}_p$), regardless of how many non-linear consumers it feeds.

XOR gates with mixed fanout

An XOR output may feed multiple AND trees with different ideal moduli. If each tree demanded its own $\mathbb{Z}_{p_i}$, the XOR result would need multiple space replicas and thus multiple switches.

Guideline. As above, evaluate all AND trees in a *single* unified $\mathbb{Z}_p$ sized for the maximum arity. Then a single switch of the XOR result from $\mathbb{B}$ to $\mathbb{Z}_p$ suffices, independent of fanout count and arity diversity.

Strategic evaluation choices for XAGs

The analyses above motivate two pragmatic assumptions that reduce total PBS usage:

(A1) AND-tree consolidation. Treat chains of two-input ANDs as a single logic unit (an $(n+1)$-input AND tree) and evaluate the entire tree with one compute PBS in a suitably sized $\mathbb{Z}_p$. This directly replaces n separate compute PBS calls with one. The required p scales linearly with the tree arity under unit-weight projection (with negacyclic pairing); in practice, we choose the smallest p that fits the LUT of the largest AND tree encountered, namely $p = 2(n+1)$.

(A2) Uniform encoding for non-linear gates. Fix a *single* plaintext space $\mathbb{Z}_p$ for *all* AND trees, where p is sized to the largest tree arity. This may lightly over-provision smaller trees, but it eliminates duplicate copies of signals across multiple $\mathbb{Z}_{p_i}$ and collapses many potential switches into *at most one* per producer.

Together, (A1) and (A2) align the encoding plan with XAG structure, reducing both compute PBS count and switch PBS count.

5.3.2 Encoding Strategy Behavior in ESOP-Based Evaluation

The ESOP form enforces (cf. the sub-subsection on "exclusive-or sum of product" in Section 2.1.1) a strict two-level structure that *maximally* separates non-linear from linear computation: the first level comprises product terms (AND trees of literals) and the second level is a single XOR tree that aggregates their outputs. This regularity is especially suitable for multi-encoding TFHE evaluation.

Implications for encoding management

The ESOP layering suggests a clean encoding plan:

- *First layer (non-linear):* evaluate each cube (i.e., each AND tree) via a compute PBS in a larger space $\mathbb{Z}_p$ sized to the largest cube arity. With unit-weight projection and negacyclic pairing showcased in Figure 5.1 b, p grows *linearly* with the maximum arity.
- *Second layer (linear):* configure each compute PBS to *emit in* $\mathbb{B}$. All cube outputs then enter the XOR aggregation layer already in $\mathbb{B}$, so the entire second layer is evaluated by free XOR, with no additional PBS.

This "compute-then-project-to-$\mathbb{B}$" rule ensures a single encoding transition per cube and *no* switches between the two layers.

Cost model for ESOP evaluation

We now quantify PBS usage under the ESOP plan above. Let:

- j_{pbs} be the number of cubes of arity ≥ 2 (each requires a compute PBS).
- $k_{\max}$ be the largest cube arity; choose p large enough to encode that cube's LUT under unit-weight projection with negacyclic pairing (linear in $k_{\max}$).
- S be the set of PIs that *must* serve both roles: they appear in at least one arity-1 cube (i.e., they must be available directly in $\mathbb{B}$) *and* also in at least one higher-arity cube (i.e., they must also be available in $\mathbb{Z}_p$).[1]

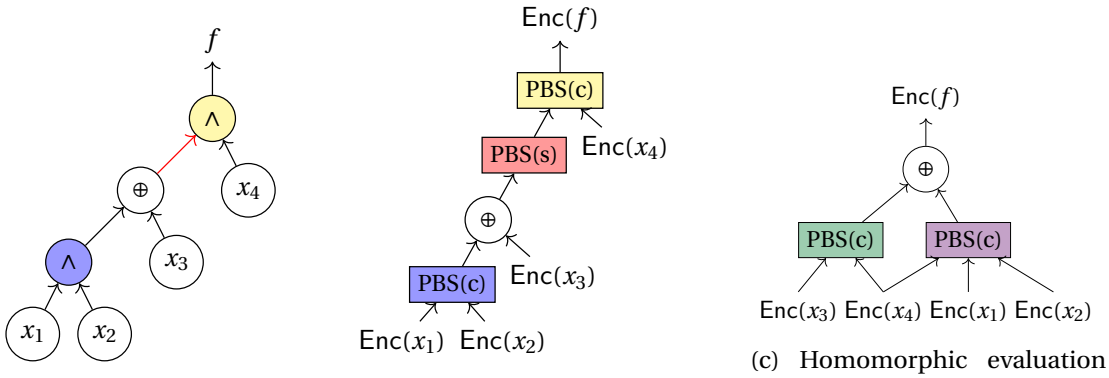
A conservative (worst-case) count of *switch PBS* is then

$$\#\text{switch PBS} \leq |S|,$$

since our plan requires at most one switch per such PI. In practice, switches of the same PI can often be coalesced, and some cubes can reuse a PI already present in $\mathbb{Z}_p$, so $|S|$ is an upper bound; our synthesis will place switches to minimize their realized count. As a result, the ESOP evaluation cost is dominated by

$$\underbrace{j_{\text{pbs}}}_{\text{compute PBS}} + \underbrace{|S|}_{\text{switch PBS (upper bound)}}.$$

[1] When a PI appears only in higher-arity cubes, it can be switched once into $\mathbb{Z}_p$ (or encrypted directly in $\mathbb{Z}_p$). When it appears only as arity-1 cubes, it remains in $\mathbb{B}$. The ambiguity — and thus the switch — arises only for PIs used in *both* roles.



(a) An MC-minimum XAG implementation of f.

(b) Homomorphic evaluation plan induced by the XAG in (a).

(c) Homomorphic evaluation plan induced by the ESOP representation $f = x_3x_4 \oplus x_1x_2x_4$.

Figure 5.3: Efficient homomorphic evaluation of function $f = ((x_1 \wedge x_2) \oplus x_3) \wedge x_4$ unlocked by its ESOP representation.

Example 5.3.1. *Consider the four-variable Boolean function* $f = ((x_1 \wedge x_2) \oplus x_3) \wedge x_4$*, whose MC-minimum XAG implementation is shown in Figure 5.3 a. An ESOP representation of the function is known to be*

$$f(x_1, x_2, x_3, x_4) = x_3x_4 \oplus x_1x_2x_4.$$

Within this ESOP, both cubes have arity ≥ 2*, so* $j_{pbs} = 2$*. The largest arity is* 3*, so* p *is chosen just large enough for that cube's LUT; specifically,* $p = 2 \times 3 = 6$*, as discussed at the end of Section 5.2.1. Since no arity-*1 *cubes are present,* $S = \varnothing$ *and no switch PBS is required; the corresponding homomorphic evaluation plan is visualized in Figure 5.3 c — the evaluation of each product term invokes a compute PBS, which is the box with the same color. In contrast, the evaluation plan following the MC-minimum XAG, shown in Figure 5.3 b, includes two compute PBS (for the two AND nodes) and one switch to reconcile the XOR/AND boundary (the arrow pointing from the XOR node to the upper AND node in Figure 5.3 a, marked in red). This example highlights the structural advantage of ESOP in facilitating efficient TFHE-based homomorphic evaluation, and motivates ESOP-guided synthesis under a multi-encoding model.*

5.4 ESOP-Guided TFHE Circuit Synthesis via Cost-Aware Cut Replacement

This section presents a synthesis methodology that transforms a Boolean circuit into an optimized network of ESOP-based sub-circuits for efficient TFHE evaluation under a multi–encoding-space model. Whereas Section 5.3 analyzed how to *evaluate* a given representation (XAG or ESOP) with minimal PBS operations, our focus here is on *generation*: how to restructure a given network so that it admits evaluation with fewer compute PBS calls and, crucially, fewer switch PBS calls. The core idea is a cut-based mapping framework that considers small, overlapping subgraphs (*cuts*), replaces them with locally optimal ESOP implementations, and commits

only those replacements that reduce a cost metric reflecting both compute and switch PBS overhead.

5.4.1 High-Level Flow

Given a Boolean circuit (represented, e.g., as an XAG), the flow proceeds node-by-node in topological order:

1. **Cut enumeration.** For each node n, enumerate all k-feasible cuts $C = (r{=}n, L)$ (Section 5.3), with at most k leaves. In practice, we use small k ($k = 5$) to balance quality and runtime.
2. **Local ESOP search.** For each cut C, derive a candidate ESOP for its cut function f_C using `exorcism` [129]. This returns an ESOP expression $\mathscr{E} = \bigoplus_{i=1}^{m} P_i$ where each cube P_i is a conjunction of literals over L.
3. **Cut-aware cost estimation.** Estimate the PBS cost of evaluating $\mathscr{E}$ *in context*, i.e., when $\mathscr{E}$ is spliced into the current partially mapped network. The estimator explicitly counts compute PBS calls (for cubes with arity > 1) and switch PBS calls (to align leaf encodings with the non-linear layer), cf. Section 5.4.2.
4. **Dynamic programming selection.** Choose, for each node, the representative cut that minimizes a global cost estimate, using a two-phase DP (area-flow passes for global sharing, followed by exact-area passes for local precision; cf. the sub-subsection on "selection" in Section 4.3.2).
5. **Network construction.** After the final pass, traverse the graph in reverse topological order and instantiate the ESOP implementation of each node's selected cut (outputting ciphertexts in $\mathbb{B}$ so that XOR aggregation remains free).

This process yields a network of ESOP sub-circuits whose structure is deliberately aligned with the dual-domain evaluation model from Section 5.3: non-linear cubes are grouped and bootstrapped in $\mathbb{Z}_p$ (with p sized for the largest cube arity), and their outputs are immediately converted (as part of the compute PBS) to $\mathbb{B}$ for free-XOR aggregation.

5.4.2 Cut-Aware ESOP Cost Modeling

When ESOP evaluation is applied to *sub-circuits* (cuts) rather than whole networks, the naïve ESOP cost model from Section 5.3.2 must be refined to reflect boundary conditions:

- **Compute PBS.** Each cube P_i with arity > 1 is evaluated by *one* compute PBS in $\mathbb{Z}_p$, where $p \geq 2 \max_i \text{arity}(P_i)$ to encode its LUT. Single-literal cubes (arity = 1) are linear and require no compute PBS.

Algorithm 5.1: Cut-Aware ESOP Cost Estimation

Input: Cut $C = (r, L)$ with cut function f_C
Output: Local PBS cost c_{local}

$\mathscr{E} \leftarrow \texttt{exorcism}(f_C)$ `// ESOP:` $\bigoplus_i P_i$
$j_{\text{pbs}} \leftarrow \#\{P_i \in \mathscr{E} \mid \text{arity}(P_i) > 1\}$
$m_{\text{clean}} \leftarrow 0$ `// leaves that never need a switch`
foreach $l \in L$ **do**
  if *l appears only in exactly one cube of arity* 1 **then**
    $m_{\text{clean}} \leftarrow m_{\text{clean}} + 1$
$s_{\text{switch}} \leftarrow |L| - m_{\text{clean}}$
return $c_{local} \leftarrow j_{pbs} + s_{switch}$

- **Switch PBS at the cut boundary.** Leaves $l \in L$ arrive in $\mathbb{B}$ (because upstream ESOPs output in $\mathbb{B}$ for XOR aggregation). If l participates in any cube of arity > 1, l must be *available* in $\mathbb{Z}_p$ at the cube's compute PBS. This alignment requires a switch PBS *unless* l appears *only* in a single arity-1 cube (i.e., it is consumed exclusively by the XOR layer).

We capture these effects with the following localized estimator (defaulting to unit weights, see the remarks that follow):

Remarks and practical refinements.

1. *Arity and feasibility.* If $\max_i \text{arity}(P_i) = k_{\max}$ exceeds the global non-linear domain capacity ($p/2$), the cut is *infeasible* for the current p and is discarded. In practice, we set $p = 2K$, where K is a user- or tool-chosen upper bound on cube arity, consistent with Section 5.3.

2. *Literal polarity.* Complemented literals do not change arity. Negations are absorbed as sign flips in the linear combination that forms the PBS input and are "free" at the TLWE level.

3. *Context-sensitivity.* The test "appears only in one arity-1 cube" is a conservative sufficient condition for being switch-free. If the same leaf is reused by multiple arity-1 cubes (across sibling cuts), a single instance can be fanned out in $\mathbb{B}$ at no PBS cost; the estimator can be extended with memoization to recognize such reuse across already committed cuts.

4. *ESOP caching.* To avoid repeated calls to `exorcism`, we cache ESOPs (and their costs) by a signature comprising the cut truth table and a canonical leaf ordering.

Algorithm 5.2: Per-Round Cut Selection (one DP pass)

Input: XAG G, prior best costs best_cost$(\cdot)$
Output: Updated representative cut rep(n) and best_cost(n) for all n

foreach node $n \in G$ in topological order **do**
  best_cost$(n) \leftarrow +\infty$; rep$(n) \leftarrow \perp$
  foreach k-feasible cut C rooted at n **do**
    $c_{\text{local}} \leftarrow$ `ESOP_cost`(C) `// Alg. 5.1`
    $c \leftarrow$ `cost_est`(C, c_{local})
    if $c <$ best_cost(n) **then**
      best_cost$(n) \leftarrow c$; rep$(n) \leftarrow C$
return rep$(\cdot)$, best_cost$(\cdot)$

5.4.3 Cost-Aware Cut Selection

Selecting a good set of replacements requires balancing local improvements with global sharing. We adopt a standard two-heuristic dynamic programming scheme (as in technology mapping, cf. Section 4.3), adapted to our cost metric.

Global cost estimators. We alternate four DP passes:

- *Area flow* in Passes 1–2: A global sharing heuristic that accumulates costs downstream, dividing each child's contribution by its fanout to reflect reuse.[2]
- *Exact area* in Passes 3–4: A local precision heuristic that measures the size of the node's maximum fanout-free cone. We simulate committing a candidate cut, recursively adding the c_{local} of any newly needed internal nodes (whose reference count rises from 0 to 1), and ignoring nodes already "paid for" by other selections. This effectively counts how many PBS calls would *actually* be added by this choice.

Commit and construction. After the final pass, each node n has a representative cut rep(n). We then build the output network by reverse-topological traversal: for each node, instantiate the ESOP of rep(n), implement each arity-> 1 cube by a compute PBS in $\mathbb{Z}_p$ (with p large enough for the largest cube arity), and set the PBS output encoding to $\mathbb{B}$ so that the XOR layer is evaluated via free-XOR. Cuts whose ESOP contains only arity-1 cubes are implemented entirely in $\mathbb{B}$ with leveled operations.

5.4.4 Practical Considerations and Implementation Notes

- **Cut size k.** Larger k exposes deeper ESOP opportunities but increases enumeration and ESOP search time. We find $k = 5$ to be a good trade-off.

[2] We use $c = \left(c_{\text{local}} + \sum_{l \in L} \text{best\_cost}(l)/\text{fanout}(l)\right)$, with PIs contributing zero.

- **Tie-breaking.** When multiple cuts have identical cost, prefer
  (i) smaller $k_{\max}$ (to keep p small), then
  (ii) fewer cubes, then
  (iii) lexicographically smaller leaf IDs (stability).
- **Global plaintext space.** Consistent with Section 5.3, we set a single $p = 2K$ for all non-linear cubes, where K is the largest cube arity *across all committed cuts*. This avoids proliferating switch PBS calls due to heterogeneous p.
- **Caching and signatures.** We cache $\langle f_C, \text{leaf\_order} \rangle \mapsto \mathscr{E}$ and $\mapsto c_{\text{local}}$ to amortize `exorcism` calls. A simple NPN-aware hash of the cut truth table suffices in practice.
- **Soundness.** Every replacement implements f_C exactly. Since cuts form a standard covering (one representative cut per node, built in reverse topological order), functional equivalence to the input circuit is preserved by construction.

5.5 Experimental Results

We evaluate the effectiveness of the proposed ESOP-guided synthesis technique for TFHE circuit generation on a suite of general-purpose Boolean benchmarks. To make the results interpretable and actionable, we structure the discussion in two parts:

1. an *evaluation-time comparison*, where we compare estimated homomorphic evaluation latency across all methods; and
2. an *analysis of PBS types*, where we separate *compute* and *switch* PBS to expose the impact of encoding strategy management on total cost.

5.5.1 Overview and Methodology

Prior work has shown that carefully hand-crafted cryptographic circuits — whose structures cleanly separate linear and non-linear logic—benefit substantially from multi-encoding-space evaluation [28]. However, outside this niche, most Boolean computations encountered in practice (analytics, control, bit-twiddling, random logic) do *not* exhibit such regularity. As a result, the broader practicality of multi-encoding evaluation depends on algorithmic synthesis support that can *create* or *recover* structure suitable for efficient encoding management.

Benchmarks. We therefore evaluate on the LOBSTER benchmark suite [112], which is also adopted in Section 3.6: medical diagnostics, sorting, bit-twiddling, random logic, and control logic (cf. Section 2.3.4). These designs are representative of computations expected in real-world FHE deployments and, unlike crypto circuits, typically *lack* deliberate separation between XOR and AND structure — making them an appropriate and challenging testbed.

Compared methods. We compare three synthesis strategies:

1. *AND minimization* [171]: reduces the number of AND gates in an XAG to lower the number of *compute PBS*. Evaluation uses the multi-encoding strategy in Section 5.3.1 (free-XOR in $\mathbb{B}$, AND trees via PBS in $\mathbb{Z}_p$), so both compute and switch PBS may occur.

2. $\mathbb{Z}_8$*-based synthesis* (Chapter 4): targets a fixed larger space $\mathbb{Z}_8$ to enable large/specialized gates and multi-output gates via multi-value PBS. Because the encoding space is fixed, all PBS are *compute* PBS.

3. *ESOP-guided synthesis* (this chapter): replaces k-feasible cuts by ESOP subcircuits using the cost-aware selection from Section 5.4. Evaluation follows the encoding choices in Section 5.3: AND-tree consolidation and a uniform $\mathbb{Z}_p$ for all non-linear cubes; XOR aggregation in $\mathbb{B}$ (free-XOR). Both compute and switch PBS may occur, but are *explicitly managed* during synthesis.

Implementation and cost estimation. All flows are implemented on top of `mockturtle` [165]. For ESOP synthesis within cuts we use `exorcism` [129]. Homomorphic evaluation latency is estimated with a TFHE cost estimator [3] that queries `Concrete` [198] to select parameters and return a per-PBS time based on the required plaintext space p and the Euclidean norm of the linear combination (scaled appropriately for multi-value PBS when relevant). XOR realized via leveled TLWE operations in $\mathbb{B}$ (*free-XOR*) is treated as negligible relative to PBS and thus excluded. Experiments run on an Apple M1 Max (32GB RAM).

Reporting. For each benchmark in Table 5.1 we report:

(i) the arity of the largest AND tree ("max arity") under the evaluation assumptions in Section 5.3.1;

(ii) the total number of PBS ("#PBS"), including both compute and switch PBS where applicable;

(iii) synthesis time ("Syn.", in seconds); and

(iv) total estimated homomorphic evaluation time ("Eval.", in milliseconds).

5.5.2 Performance Breakdown

Evaluation time comparison. No single method dominates across all 25 benchmarks. The best evaluation time is achieved by *AND minimization*, $\mathbb{Z}_8$*-based synthesis*, and *ESOP-guided*

[3] https://github.com/ssmiler/tfhe_lbf_eval

Benchmark	*AND Minimization*				*$\mathbb{Z}_8$-Based Synthesis*			*This Work (ESOP-guided)*			
	Max arity	#PBS	Syn. [s]	Eval. [ms]	#PBS	Syn. [s]	Eval. [ms]	Max arity	#PBS	Syn. [s]	Eval. [ms]
cardio	12	141	24.14	6 139	114	<0.01	4 560	8	120	5.98	4 392
dsort	5	789	564.00	29 316	663	<0.01	26 520	5	828	31.39	31 617
msort	5	690	50.37	23 715	645	<0.01	25 800	5	528	100.88	19 602
isort	5	690	50.17	23 715	645	<0.01	25 800	5	528	100.89	19 602
bsort	5	690	51.47	23 715	645	<0.01	25 800	5	528	100.64	19 602
osort	5	598	76.45	20 553	559	<0.01	22 360	5	460	83.38	17 058
hd01	32	64	0.43	5 096	54	<0.01	2 160	5	59	0.58	2 296
hd02	32	62	0.60	4 030	78	<0.01	3 120	5	81	1.33	2 791
hd03	4	51	0.44	1 679	29	<0.01	1 160	4	52	1.13	1 748
hd04	26	59	7.86	3 298	53	<0.01	2 120	7	65	2.28	2 500
hd05	17	191	20.70	9 173	113	<0.01	4 520	6	153	2.00	5 725
hd06	17	191	21.67	9 173	113	<0.01	4 520	6	153	2.02	5 725
hd07	4	15	0.05	555	16	<0.01	640	4	14	0.08	526
hd08	13	20	0.09	874	11	<0.01	440	8	12	0.17	508
hd09	33	69	1.02	4 521	76	<0.01	3 040	8	108	1.73	3 964
hd10	32	6	0.08	534	25	<0.01	1 000	8	16	0.27	624
hd11	48	168	8.37	23 397	310	<0.01	12 400	8	411	12.71	15 391
hd12	32	67	0.42	3 887	73	0.01	2 920	8	59	2.05	2 287
bar	3	1 980	25.37	61 887	2 437	<0.01	97 480	3	2 176	51.62	68 480
cavlc	30	365	19.12	22 300	518	<0.01	20 720	8	335	7.83	14 371
ctrl	10	61	1.85	2 605	84	<0.01	3 360	7	73	0.36	3 047
dec	8	280	25.37	12 280	292	<0.01	11 680	2	352	2.49	10 208
i2c	26	535	24.11	40 217	890	<0.01	35 600	8	703	10.07	29 171
int2float	14	127	5.91	6 665	162	<0.01	6 480	8	117	2.03	4 849
router	70	51	7.78	7 639	111	<0.01	4 440	8	95	1.51	3 555
GEOMEAN		141.32		7 350.23	152.00		6 079.87		150.22		5 634.95
Norm. (Eval.)				**1**			**0.8272**				**0.7666**

Table 5.1: Comparison of TFHE circuit evaluation costs concerning the LOBSTER benchmark suite.

synthesis on 2, 9, and 14 benchmarks, respectively. This supports the perspective that the three methods exploit different forms of structure: $\mathbb{Z}_8$-based synthesis excels on bit-twiddling (hd-01–hd-12), where many gates are compressible and multi-output opportunities abound in a fixed, larger domain; ESOP-guided synthesis wins broadly elsewhere (medical, control, mixed random logic), where its cut-wise restructuring can regularize non-linear parts and reduce encoding friction.

Circuit synthesis time. $\mathbb{Z}_8$-based synthesis is consistently fastest to *synthesize* due to its technology-mapping nature with a static library. *AND minimization* and *ESOP-guided* exhibit comparable synthesis times (often minutes): the former because aggressive multi-pass optimization converges slowly on complex graphs; the latter because it performs ESOP discovery per cut. The latter overhead can be reduced in practice via memoization or a precomputed ESOP database keyed by cut truth tables and simple invariants. While we report synthesis time for completeness, in many deployments (e.g., cloud services) circuit generation is an offline, amortized cost; homomorphic evaluation latency is the dominant metric.

Cross–family perspective (leveled vs. fast-bootstrapping). Because this chapter reuses exactly the same 25 benchmarks as our leveled-FHE study in Chapter 3, it is natural to contrast

Benchmark	*AND Minimization*		*This Work (ESOP-guided)*	
	#Switch	#Compute	#Switch	#Compute
`cardio`	91	50	63	57
`dsort`	294	495	243	585
`msort`	405	285	198	330
`isort`	405	285	198	330
`bsort`	405	285	198	330
`osort`	351	247	174	286
`hd01`	19	45	14	45
`hd02`	31	31	47	34
`hd03`	31	20	28	24
`hd04`	36	23	24	41
`hd05`	112	79	61	92
`hd06`	112	79	61	92
`hd07`	3	12	2	12
`hd08`	13	7	2	10
`hd09`	34	35	56	52
`hd10`	1	5	6	10
`hd11`	73	95	194	217
`hd12`	40	27	23	36
`bar`	491	1 489	384	1792
`cavlc`	200	165	44	291
`ctrl`	23	38	11	62
`dec`	20	260	48	304
`i2c`	177	358	154	549
`int2float`	56	71	26	91
`router`	31	20	45	50
GEOMEAN	59.93	71.92	45.78	97.44
Norm.	**1**	**1**	**0.7638**	**1.3548**

Table 5.2: Breakdown of PBS usage: compute vs. switch, for flows that support multi-encoding evaluation.

the end-to-end homomorphic evaluation efficiency across the two major families of schemes. On that suite, the best-performing leveled-FHE flow from Chapter 3 — our framework with the FHE cost configured as the optimization objective — achieved a geometric-mean evaluation time of 18 198.80ms (cf. Table 3.3). In comparison, on the very same benchmarks, the ESOP-guided dual-space TFHE flow developed in this chapter attains a geometric mean of 5 634.95ms (Table 5.1), i.e., about 3.23× faster overall. While these numbers come from different software stacks and measurement methodologies (Chapter 3 reports measured runtimes in `HElib`, whereas this chapter uses a TFHE cost estimator backed by `Concrete`), they nonetheless provide a clear, broad-strokes picture: fast-bootstrapping schemes such as TFHE are particularly well-suited to the private evaluation of functions over small plaintext domains — especially Boolean logic.

5.5.3 Analysis of PBS Types

To isolate the impact of encoding management, we break the total PBS into *compute* and *switch* components for *AND minimization* and *ESOP-guided* (Table 5.2). The $\mathbb{Z}_8$-based flow is excluded here because its encoding space is fixed and all PBS are compute PBS.

On average, ESOP-guided synthesis reduces *switch* PBS by 23.62% versus AND minimization.

This is the direct benefit of encoding-aware, ESOP-structured subcircuits: inputs that cross the $\mathbb{B} \leftrightarrow \mathbb{Z}_p$ boundary are minimized and aligned with ESOP layer boundaries (Section 5.3), so transitions are fewer and better placed.

As expected, ESOP-guided shows a 35.48% increase in *compute* PBS relative to AND minimization, which explicitly targets AND-count reduction. However, the net effect still favors ESOP-guided in total latency (Table 5.1), because:

(i) fewer switch PBS means fewer "overhead-only" bootstraps; and

(ii) ESOP-guided avoids extremely large AND trees, enabling smaller plaintext spaces and lower per-PBS cost.

In short, while both flows support multi-encoding evaluation, only ESOP-guided *manages* encoding transitions during synthesis. That management produces circuits whose PBS mix and p selection are better balanced, yielding lower end-to-end homomorphic evaluation time.

5.6 Discussion

This chapter studied *encoding–switch–aware* TFHE circuit synthesis: when and how to change plaintext spaces during evaluation, and how to structure circuits so that such changes are both necessary and beneficial. This chapter summarized the mechanics of PBS under multiple plaintext spaces, analyzed where switches arise in XAG and ESOP representations (Section 5.3), and then designed a cost-aware, cut-based replacement flow that carves a network into ESOP-aligned regions while explicitly accounting for both *compute* and *switch* PBS operations (Section 5.4). On 25 general-purpose benchmarks, the resulting circuits reduce homomorphic evaluation time by up to 53.46% and on average by 23.34% compared to advanced baselines that do not explicitly manage encoding transitions (Section 5.5). Below is a reflection on what the results say and the outlined directions that are believed to be the most promising.

5.6.1 What this chapter contributes

- **A principled view of switching.** *Compute PBS* (logic evaluation) is distinguished from *switch PBS* (space transition only). Their roles in interacting with circuit topology are shown. In XAGs, AND-tree consolidation and uniform non-linear encoding reduce avoidable switches; in ESOPs, the two-level structure provides a clean separation that naturally minimizes switching.
- **A localized yet context-aware cost model.** Our cut-level estimator (Algorithm 5.1) counts the compute PBS induced by higher-arity cubes and the switch PBS implied at cut boundaries, enabling fine-grained trade-offs during mapping.
- **A synthesis flow that *uses* switching — not just *allows* it.** By selecting ESOP implemen-

tations cut-by-cut with dynamic-programming style selection, the flow actively shapes where logic is computed (in $\mathbb{Z}_p$) and where it is aggregated (in $\mathbb{B}$), rather than applying a fixed-space policy or switching opportunistically post hoc.

- **Evidence across diverse, non-cryptographic circuits.** Beyond structured cryptographic designs, our study shows that general Boolean workloads benefit from encoding-aware synthesis, often by controlling cube arities (hence p) and by eliminating gratuitous switches created by mixed-fanout substructures.

5.6.2 Limitations

Our evaluation times are derived from a TFHE cost estimator backed by `Concrete`; absolute numbers will vary with libraries and hardware, though the *relative* trends we exploit — controlling p via cube arity and reducing switch PBS — are robust. Also, our flow currently synthesizes ESOP per cut on the fly; for large designs, ESOP generation dominates runtime.

5.6.3 Future directions

(1) Matching applications to evaluation strategies. Our results indicate that no single strategy dominates all benchmarks. Fixed large-space mapping (Chapter 4) and the ESOP-guided flow in this chapter each win on different classes of functions. A compelling next step is to *predict* which strategy best fits a given Boolean function — or even a region within a function. Concretely:

- *Structural features.* Use features such as XOR/AND ratio, AND-tree arity histogram, reconvergence patterns, ESOP cube-size distribution, and MFFC statistics to characterize a circuit.
- *Cost surrogates.* Learn a regressor that maps these features to estimated (compute, switch) PBS counts and to the implied p and N parameters, yielding a quick ranking of strategies before full synthesis.
- *Per-region decisions.* Extend the predictor to operate on cuts or partitions, enabling a mixed-strategy design within one circuit while keeping switch overhead explicit in the objective.

The outcome would be a *portfolio* synthesizer that selects among (or blends) strategies based on circuit semantics, rather than committing globally a priori.

(2) Toward *heterogeneous* TFHE synthesis with controlled interfaces. This chapter and Chapter 4 take complementary angles. Chapter 4 maximizes expressiveness within a *single* (larger) plaintext space by technology mapping onto large, compression-friendly gates

(e.g., symmetric/negacyclic) and multi-value PBS; this chapter optimizes *when/how* to switch spaces, using ESOP to keep non-linear blocks small (thereby keeping p modest) and to minimize switches. Combining the two is non-trivial because our ESOP-guided flow already fixes both the switching boundary (between non-linear and XOR layers) and the target p (by the largest cube), leaving little room to invoke large-gate libraries inside those ESOP regions.

A plausible path forward is **heterogeneous synthesis**: allow the mapper to designate some regions as *ESOP domains* (dual-space, free-XOR aggregation) and others as *large-gate domains* (fixed large p, technology mapping per Chapter 4). To make this viable:

- *Region selection.* Identify ESOP regions with *low logic sharing* (where multi-output advantages are limited) and high switch pressure — these are good candidates to keep in the ESOP domain. Conversely, detect regions rich in symmetric/negacyclic structure or high fan-in opportunities with shared supports — prime candidates for large-gate domains with MV-PBS.
- *Interface contracts.* Treat domain boundaries as typed interfaces: each crossing incurs a switch PBS (if needed) and may constrain the output space ($\mathbb{B}$ vs. $\mathbb{Z}_p$). The global objective must price these gates explicitly.
- *Joint cost model.* Extend our per-cut estimator to a bi-domain model that

  (i) prices large-gate primitives (including MV PBS amortization) and

  (ii) retains switch PBS accounting between domains.

  A dynamic programming/integer linear programming hybrid could then select domain types per cut while ensuring acyclic coverage.

The goal is not to indiscriminately combine techniques, but to *partition* the circuit so each region is synthesized by the method that best matches its structure, with switching overhead controlled at the seams.

(3) Engineering opportunities. Finally, two practical improvements are immediate:

(i) cache or precompute a library of high-quality ESOPs (by support and polarity) to reduce per-cut synthesis time; and

(ii) parallelize cut evaluation and cost estimation, which are embarrassingly parallel across nodes.

In summary, this chapter establishes a foundation for *systematic* encoding strategy management in TFHE synthesis: analyzing where switches arise, costing them alongside compute PBS, and using those insights to restructure circuits into encoding-aligned ESOP regions. The

results, together with Chapter 4, suggest a broader agenda: *structure-aware* TFHE design automation that chooses *both* the operators *and* the encoding spaces best suited to the logic at hand.

Part II

Toward Practical Garbled Circuits: Logic-Level Innovations for Garbled Boolean Circuits

6 Ciphertext-Efficient Garbled Circuits via XOR-OneHot Graphs

6.1 Motivation

Why garbled circuits, and why now. *Garbled circuits* (GC) remain one of the most versatile protocols for general-purpose MPC: an arbitrary Boolean circuit is transformed into a network of encrypted tables and evaluated without revealing intermediate values [190]. Compared with FHE-based MPC protocols [136, 135], GC avoids heavy noise management and typically involves lighter computational overhead; compared with *secret sharing* [159], it protects intermediate results by design. These properties make GC attractive for privacy-preserving analytics, secure model inference, and sensitive-data collaboration in domains such as finance [27, 60, 85] and healthcare [50].

The bottleneck: garbling cost (communication). Despite its simplicity and robustness, GC performance is dominated by *garbling cost* — the number of ciphertexts that must be communicated. Modern schemes have slashed the per-gate cost: *Free-XOR* renders XOR gates free (no ciphertexts required for a secure gate evaluation) by exploiting a global key offset [108]; *half-gates* garble any two-input nonlinear gate (e.g., AND) with only two ciphertexts [197]; and *garbling gadgets* show that any n-input *symmetric* gate can be garbled with at most n ciphertexts [16] (see Section 6.2.1 for a brief introduction). Yet, real applications remain communication-bound, so further progress hinges on *logic-level* techniques that minimize the #ciphertexts a circuit induces.

Status quo: minimize ANDs in XAGs. The dominant logic-level approach models the computation as an *XOR–AND graph* (XAG) and minimizes the number of two-input ANDs, since XOR/NOT are free under free-XOR [166, 152, 171, 118]. This "MC reduction" viewpoint (minimizing multiplicative complexity) has delivered strong baselines. However, it bakes in a strong assumption: that the *two-input AND* is the best nonlinear primitive everywhere. With half-gates (2 ciphertexts per two-input AND gate) and free-XOR, this is often — but not always — optimal.

Our premise: ciphertext efficiency varies across primitives. We revisit that assumption by asking — "Is there a circuit representation whose nonlinear primitives yield **lower total ciphertexts** than XAGs, once free-XOR and modern garbling schemes are fully exploited?" Starting from a systematic study of ciphertext efficiency across small nonlinear gates under state-of-the-art garbling, we observe that certain *three-input symmetric* primitives — especially the *OneHot* gate, which outputs 1 iff exactly one input is 1 — can be strikingly efficient when combined with aggressive linear (XOR) restructuring. In our truth-table notation, the OneHot gate is $^{\#}16$ in hexadecimal and $^{\#}00010110$ in binary. While a single OneHot costs 3 ciphertexts according to the garbling gadget technique [16], it can *replace two two-input AND gates at once*, whose garbling requires 4 ciphertexts. The net result can be a lower *circuit-wide* ciphertext count than an AND-only nonlinear basis.

From XAG to X1G: a representation built for GC. Motivated by this finding, we introduce the *XOR–OneHot-inverter graph* (X1G): a Boolean network whose nonlinear primitives are *OneHot* gates, while XOR/NOT remain free. X1Gs explicitly trade a slightly larger per-gate cost (e.g., 3 for a OneHot gate vs. 2 for a two-input AND) for *fewer nonlinear sites overall* and substantially more linear reuse. The representation is tailored to modern garbling: XOR-heavy structure is "free," while the remaining nonlinearity is delivered by primitives with superior *ciphertext-per-functionality* in many common patterns, as will be revealed later in this chapter.

What it takes to make X1G practical. To translate this premise into practice, one must

(i) determine *when* OneHot-based nonlinearity beats AND-based nonlinearity,

(ii) *map* Boolean functions to X1Gs that realize those wins,

(iii) *optimize* X1Gs aggressively to exploit logic sharing, in order to obtain ciphertext-efficient garbled circuit designs.

This work contributes the following:

- **Ciphertext-efficiency analysis.** We characterize, for the first time, the garbling cost behavior of small symmetric gates under free-XOR, half-gates, and garbling gadgets, identifying regimes where OneHot primitives dominate two-input ANDs.
- **Exact X1G synthesis for small functions.** We formulate and solve an exact synthesis problem that produces *garbling-cost-optimal* X1Gs for small Boolean functions, enabling a reusable library.
- **Heuristic X1G optimization for practical functions.** We design a suite of X1G-native optimization passes (linearization, factoring, fuse/split of OneHots, and gadget-aware rewriting) that scale to realistic circuits and preserve GC-friendliness at every step.

- **Robust empirical gains.** Across diverse benchmark suites, our X1G flow reduces total garbling cost by 7.34%, 26.14%, 13.51%, and 4.34% on average, respectively, compared with state-of-the-art MC-reduction pipelines, at acceptable runtime overheads.

Scope and positioning. This chapter focuses on *communication-efficient* SMPC via GC and is in parallel to the previous chapters on FHE (cf. Part I). Here, we optimize GC's *communication bound* by rethinking the circuit representation itself. The proposed X1G framework is with the mainstream cryptographic constructions integrated (free-XOR, half-gates, garbling gadgets) and *orthogonal* to cryptographic improvements — making it immediately compatible with current GC engines and future garbling advances.

The results and techniques presented in this chapter are drawn from our earlier publication in [193, 195, 194].

6.2 Preliminaries

We defer the introduction of the garbling gadget technique until this point, as our later analysis of the *MC compactness* of Boolean operators (see Section 6.3.1) critically depends on it. For background on the GC protocol and other advanced garbling techniques, such as free-XOR and half-gates, please refer to Sections 2.2.3–2.2.4.

6.2.1 Garbling Gadget

A straightforward garbling of an n-input gate requires a garbled table with 2^n ciphertexts — one per truth-table row — so large fan-in gates are typically avoided when synthesizing GC circuits. *Garbling gadget* [16] is an advanced garbling technique that shows this pessimism is unnecessary for a broad and important class of gates: *symmetric functions* (whose output depends only on the Hamming weight of the inputs) can be garbled with *linear*, rather than exponential, cost.

Idea in a nutshell. Let $f : \{0,1\}^n \mapsto \{0,1\}$ be symmetric. Then there exists a predicate $g : \{0,1,\ldots,n\} \mapsto \{0,1\}$ such that

$$f(x_1,\ldots,x_n) = g\Big(\sum_{i=1}^{n} x_i\Big).$$

Choose a modulus $z \leq n+1$ large enough that g is well-defined on $\mathbb{Z}_z$. The garbling gadget evaluates f in two stages:

1. **Modular accumulation (linear stage).** Compute the sum $s = \sum_i x_i \mod z$. This is a linear computation over $\mathbb{Z}_z$, so it can be realized in GC using encodings compatible with

free-XOR — no ciphertexts are needed for this stage.

2. **Projection (nonlinear stage).** Map the residue $s \in \{0,\dots,z-1\}$ to the output bit via a small "z-to-2" *projection gadget* that implements g. Using *garbled row reduction (GRR3)* [139], this table needs only $z-1$ ciphertexts.

Overall, an n-input symmetric gate is garbled with at most $z-1 \le n$ ciphertexts, a dramatic improvement over the naïve 2^n.

An Example. When exploiting a garbling gadget to garble a two-input AND: since $z=3$, the projection costs $z-1=2$ ciphertexts, which matches the *half-gates* optimum for two-input AND [197].

The key takeaway is conceptual: *large fan-in symmetric gates can be more garbling-efficient nonlinearity providers than two-input ANDs.* This conjecture motivates logic-level representations that surface symmetric structure so it can be garbled with near-linear cost — an idea we will exploit later when proposing circuit forms alternative to XAG for low-cost GC generation.

6.3 A Ciphertext-Efficient Logic Representation

According to the definition of *functional multiplicative complexity* (FMC) in Section 2.1.1, a two-input AND (AND2) is the most immediate nonlinearity provider in XAGs. However, AND2 is not necessarily the *most ciphertext-efficient* primitive under modern garbling schemes. This section asks a sharper question: *Is there a logic primitive that delivers the same (or more) nonlinearity for fewer ciphertexts than* AND2*?*

Motivated by the *garbling gadget* perspective (Section 6.2.1), which shows symmetric gates can be garbled far more efficiently than the naive 2^n-entry tables would suggest, we focus our search on *three-input symmetric* gates. Larger fan-in primitives ($n>3$) are excluded to keep logic synthesis practical, and asymmetric three-input gates are dominated by garbling cost.

6.3.1 MC Compactness

We quantify the "nonlinearity per ciphertext" of a gate g via *MC compactness*:

$$\text{MC-compactness}(g) = \frac{\text{functional MC of } g}{\#\text{ ciphertexts to garble } g}.$$

Higher values indicate that a gate provides more nonlinearity for each transmitted ciphertext [1].

Among the $2^{2^3}=256$ three-input Boolean functions, exactly $2^4=16$ are symmetric (the output depends only on Hamming weight). After removing the two constants and canonicalizing the

[1] Throughout, "ciphertexts" refers to entries in the garbled table that must be communicated.

Gate	Truth table	Functional MC	Garbling cost (#ciphertexts)	MC compactness
AND3	$^{\#}80$	2	3	0.67
MAJORITY	$^{\#}e8$	1	3	0.33
ONEHOT	$^{\#}16$	2	2	1.00
GAMBLE	$^{\#}81$	1	2	0.50

Table 6.1: Profile of the three-input symmetric candidates (truth tables in hexadecimal; garbling gadget cost $(z-1)$ and MC compactness.

remaining 14 by input/output negations (cf. the sub-subsection on "Boolean classification" in Section 2.1.2), five representatives remain [122]: three-input AND AND3, three-input XOR XOR3, three-input majority MAJORITY, three-input onehot ONEHOT, and three-input gamble GAMBLE. XOR3 has functional MC 0 and thus is not a nonlinearity provider; we exclude it.

Two observations stand out. First, ONEHOT and GAMBLE admit modulus $z = 3$ (garbling cost 2), whereas AND3 and MAJORITY require $z \geq 4$ (cost ≥ 3). For ONEHOT, if HMW(i) denotes all three-bit patterns of Hamming weight i, then HMW(0) = $\{0,0,0\}$ and HMW(3) = $\{1,1,1\}$ both map to output 0, so the function depends only on the Hamming weight modulo 3. The same holds for GAMBLE. Second, because ONEHOT also has functional MC 2, it achieves *unit* MC compactness (2 nonlinearity units / 4 ciphertexts) — the best among the candidates. This strongly suggests elevating ONEHOT to a first-class nonlinearity provider.

X1G: a ciphertext-efficient representation. We therefore propose *XOR–OneHot graphs* (X1Gs): logic networks over the basis {XOR, NOT, ONEHOT}. Since XOR/NOT are free under free-XOR, ONEHOT becomes the *only* source of garbling cost. Optimizing an X1G thus reduces to minimizing its number of ONEHOT nodes.

6.3.2 Properties of ONEHOT and Immediate Consequences

Only ONEHOT carries cost. Because $\mathrm{NOT}(x) = \mathrm{XOR}(1, x)$, NOT incurs no ciphertexts under free-XOR. Hence, ONEHOT is the sole cost-bearing primitive in X1Gs. We define the *X1G optimization problem* as:

> **minimize** #ONEHOT subject to functional equivalence.

Algebraic normal form (ANF) and a key identity. The ANF of ONEHOT is

$$\mathrm{ONEHOT}(x_1, x_2, x_3) = x_1 x_2 x_3 \oplus x_1 \oplus x_2 \oplus x_3. \tag{6.1}$$

A structural-MC-optimal XAG realizing ONEHOT with two AND2s is shown in Figure 6.1.

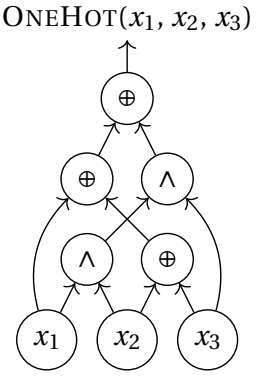


Figure 6.1: Structural-MC-optimal XAG for ONEHOT (functional MC = 2).

From Eq. (6.1) we obtain

$$\mathrm{AND3}(x_1, x_2, x_3) = \textsc{OneHot}(x_1, x_2, x_3) \oplus \mathrm{XOR3}(x_1, x_2, x_3), \tag{6.2}$$

i.e., one ONEHOT plus (free) XORs realizes AND3. Under free-XOR and garbling gadget, Eq. (6.1) reduces the cost of an AND3 from 3 ciphertexts (direct garbling with $z = 4$) to 2 (one ONEHOT with $z = 3$). This is visualized in Figure 6.2.

Useful constant-propagation rules. Fixing one input yields cost-reducing rewrites that we apply as a post-processing pass:

$$\textsc{OneHot}(1, x_1, x_2) = \neg x_1 \wedge \neg x_2 = \mathrm{AND2}(\neg x_1, \neg x_2) \quad \text{(a NOR of } x_1, x_2\text{)}, \tag{6.3}$$

$$\textsc{OneHot}(0, x_1, x_2) = x_1 \oplus x_2 = \mathrm{XOR2}(x_1, x_2). \tag{6.4}$$

Both rules replace a cost-bearing ONEHOT with cost-free primitives under free-XOR, directly lowering garbling cost.

Takeaway. Among compact three-input symmetric gates, ONEHOT uniquely combines

(i) low garbling cost ($z = 3 \Rightarrow 2$ ciphertexts),

(ii) nontrivial functional MC (2), and

(iii) favorable algebraic identities that let it stand in for other nonlinearities (e.g., AND3) using only free XOR around it.

These properties justify X1G as a ciphertext-efficient logic representation for GC generation; the remainder of this chapter develops exact and heuristic flows to minimize #ONEHOT while preserving functionality.

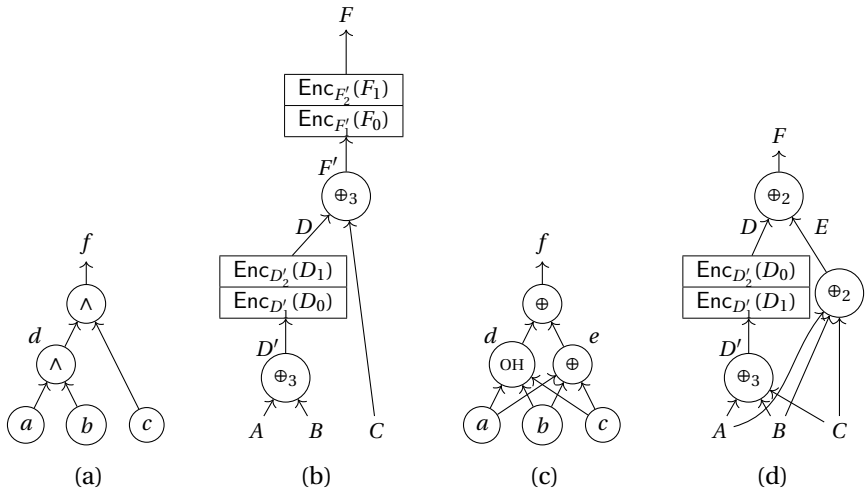


Figure 6.2: A wise choice of logic representation can lead to lower-cost garbled circuits, using the function $f = a \wedge b \wedge c$ as an example: (a) XAG implementation of f; (b) Garbled circuit corresponding to (a); (c) X1G implementation of f; (d) Garbled circuit corresponding to (d). "$\wedge$", "$\oplus$", and "OH" indicate Boolean AND, Boolean XOR, and Boolean OneHot operations, respectively. "$\oplus_z$" indicates bit-wise addition in modulo z.

6.4 Mapping XAGs to X1Gs via Algebraic Rewriting

Most prior GC-oriented logic flows implement the target function as an XAG and then reduce its structural MC (i.e., the number of AND2 gates). To benefit from that body of work *and* from the ciphertext–efficiency of ONEHOT gates (Section 6.3), we develop a lightweight mapping from XAGs to X1Gs that preserves functionality while lowering garbling cost whenever possible.

6.4.1 An Algebraic–Rewriting–Based Mapping

The key identity revealed by Eq. (6.2) allows us to replace *two adjacent* AND2s feeding a common output by a single ONEHOT plus free XORs, saving *two* ciphertexts (from four down to two) under garbling gadget (with $z = 3$) and free-XOR. However, this rule must be applied carefully: if the output of the lower AND2 has fanout > 1, removing it would disrupt other downstream dependencies.

To ensure correctness, we first *cover* the XAG to form disjoint *AND trees* whose internal nodes have unit fanout; we use the linear-time covering procedure of [131]. Inside each covered AND tree, adjacent AND2s can be merged without violating external functionality.

For any remaining AND2 that cannot be paired inside its tree, we fall back to Eq. (6.3), which preserves cost (two ciphertexts) while moving the nonlinearity to a ONEHOT gate — beneficial for downstream X1G-specific optimizations.

Algorithm 6.1: XAG→X1G mapping via algebraic rewriting

Input: XAG N
Output: Functionally equivalent X1G
// Group inner-fanout-1 AND nodes into AND trees
covering(N) // [131]
foreach AND tree root $n \in N$ **do**
  $\mathscr{M} \leftarrow$ linearize n's tree into a list of consecutive AND2s (leaf→root)
  while $\mathscr{M} \neq \emptyset$ **do**
    if $|\mathscr{M}| = 1$ **then**
      let $\{x_1, x_2\} \leftarrow$ fanins of $\mathscr{M}[0]$
      $n' \leftarrow \mathrm{OneHot}(1, \neg x_1, \neg x_2)$ // apply Eq. (6.3)
      replace node $\mathscr{M}[0]$ by n'
      remove $\mathscr{M}[0]$
    else
      let $\{x_1, x_2\} \leftarrow$ fanins of $\mathscr{M}[0]$
      $x_3 \leftarrow$ the non-$\mathscr{M}[0]$ fanin of $\mathscr{M}[1]$
      $n' \leftarrow \mathrm{XOR2}\big(\mathrm{OneHot}(x_1, x_2, x_3),\ \mathrm{XOR3}(x_1, x_2, x_3)\big)$ // apply Eq. (6.2)
      replace $\mathscr{M}[0]$ and $\mathscr{M}[1]$ by n'
      remove them from $\mathscr{M}$
return N

Complexity. After one linear-time covering pass, each AND2 is visited at most once; each visit induces $\mathcal{O}(1)$ work. The overall mapping therefore runs in time linear in the gate count of the input XAG.

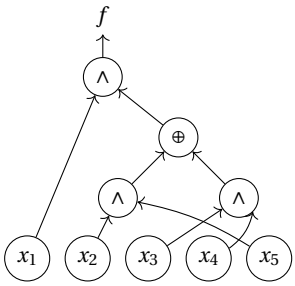
6.4.2 Limitations and a Motivating Example

Algorithm 6.1 is a fast, practical bridge from *AND-count–optimized* XAGs to lower-cost X1Gs. Yet, it can be sub-optimal when the AND connectivity in the input XAG is unfavorable. The reason is structural: in XAG optimization for MC reduction, all AND2s are "equal cost"; but after switching to X1G, *paired* AND2s (eligible for Eq. (6.2)) are preferable to *isolated* ones (which force (6.3)). Consequently, an XAG that minimizes the *count* of AND2s may lack enough adjacent pairs to realize the best X1G.

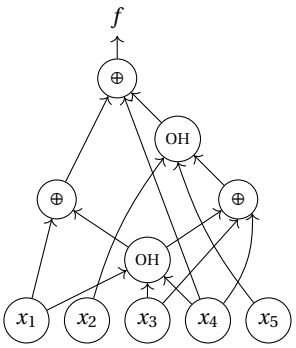
Case study. Consider the 5-variable function with truth table $^{\#}2888a000$. Its functional MC is known to be three. Figure 6.3 shows two ciphertext-optimal [2] implementations:

(a) An XAG whose structural MC matches the lower bound (3 AND2s). Mapping it with Algorithm 6.1 yields an X1G with *three* ONEHOTs (cost 6 ciphertexts).

[2] When emphasizing fewer ciphertexts as the objective, we use the term *ciphertext-optimal* rather than *optimum*: while ONEHOT is empirically the most ciphertext-efficient nonlinearity among the three-input symmetric gates we studied, proving global optimality of the representation remains open.



(a) Ciphertext-optimal XAG (structural MC= 3). Mapping via Alg. 6.1 ⇒ X1G with 3 ONEHOTs.



(b) Ciphertext-optimal X1G (two ONEHOTs). Requires an XAG with two adjacent AND2 pairs (structural MC= 4).

Figure 6.3: Ciphertext-optimal implementations for $^{\#}2888a000$. Minimizing #AND2s in XAG does not necessarily minimize ciphertexts after X1G mapping; the *pairability* of AND2s (eligibility for Eq. (6.2)) is crucial.

(b) An X1G of cost *four* ciphertexts (two ONEHOTs). The XAG that maps *into* (b) necessarily contains two *adjacent* AND2 pairs and thus has structural MC 4 — worse for XAG, yet better for the final X1G.

Thus, optimizing only the AND2 *count* in the starting XAG can paint us into a corner for X1G mapping; AND *connectivity* also matters.

Takeaway. Algebraic mapping is a near-zero–overhead way to harvest ONEHOT efficiency from existing MC-reduced XAGs, but it is not a silver bullet: to approach ciphertext optimality, the *number and* the *arrangement* of ANDs in the seed XAG must be considered. This motivates the dedicated exact and heuristic X1G-oriented synthesis/optimization engines developed in the next sections.

6.5 Agilely Synthesizing Optimal X1G Implementations for Small-Scale Functions

This section develops an exact (SAT–based) synthesis procedure that, for a given Boolean function, constructs an XAG whose structure is tailored so that the algebraic mapping of Algorithm 6.1 yields a ciphertext–optimal X1G. Unlike conventional MC-reduction flows that only minimize the #AND2s, our formulation jointly captures

(i) the *count* of AND2s, and

(ii) their *connectivity* (i.e., how many “adjacent pairs” exist),

because the latter directly controls how many OneHot substitutions of Eq. (6.2) are possible after mapping.

Exact synthesis does not scale to large functions; nevertheless, our formulation is efficient enough to solve *all* 150357 representatives of the up-to-six-variable spectral-equivalence classes. The resulting optimal implementations serve as building blocks in our database-driven optimization for practical (large) circuits.

6.5.1 AND Fence

We introduce the *AND fence* as a compact summary of how AND2s are *arranged* in an XAG [3]. The construction is:

1. Run the linear-time *covering* procedure of [131] to partition AND2s into disjoint *AND trees* (internal nodes have unit fanout). Let the i-th tree contain $c_i \geq 1$ AND2s.
2. The multiset $\mathscr{F} = \{c_1, c_2, \ldots, c_d\}$ is the *AND fence*. We order trees by network topology to make $\mathscr{F}$ unique for a given XAG.

From $\mathscr{F}$ we can read:

$$smc(\mathscr{F}) = \sum_{i=1}^{d} c_i, \tag{6.5}$$

$$cost(\mathscr{F}) = \sum_{i=1}^{d} 2 \cdot \left\lceil \frac{c_i}{2} \right\rceil. \tag{6.6}$$

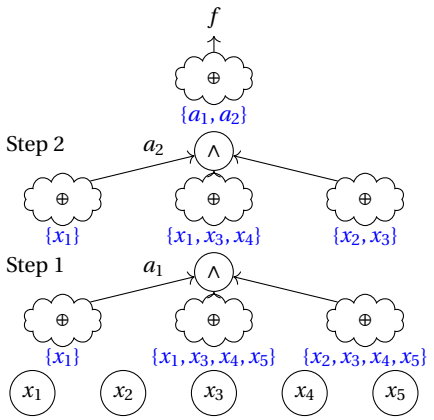
Here, (6.5) is the structural MC (the #AND2s). Equation (6.6) is the ciphertext cost of the mapped X1G under Algorithm 6.1: each pair of adjacent AND2s in a tree contributes one OneHot (two ciphertexts), and a leftover single AND2 also maps to one OneHot (two ciphertexts).

6.5.2 Abstract XAG

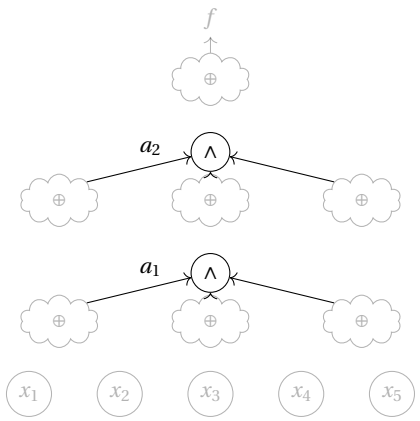
To expose just the structure that matters for AND fences, we work with a generalized *abstract XAG* [39, 163]:

(i) XOR nodes are *XOR clouds* with arbitrary fanin;

[3] We reuse the terms "AND fence" and "abstract XAG" introduced in Section 3.3.1, but tailor their definitions to a different optimization objective. In Chapter 3, the quality of an XAG was judged by *multiplicative depth* (MD) and *multiplicative complexity* (MC). Here, our goal is ciphertext cost, driven by MC and *AND connectivity*. Consequently, patterns that are indistinguishable under MD/MC — e.g., two consecutive ANDs versus two ANDs separated by an intervening XOR — are distinguished here because only the former permits a OneHot substitution via Eq. (6.2). Conversely, the present AND-fence abstraction does not attempt to distinguish all structures that affect MD, so it should not be viewed as a strict refinement of the Chapter 3 notion.



(a) Abstract XAG for $^{\#}2888a000$. Blue text denotes XOR-cloud fanins.



(b) Corresponding AND fence $\mathscr{F} = \{2,2\}$.

Figure 6.4: Abstract XAG of the ciphertext-optimal implementations for $^{\#}2888a000$ and its AND fence.

(ii) each fanin of an XOR cloud is either a PI or an AND node from a lower level;

(iii) each AND node takes XOR clouds as inputs;

(iv) POs are XOR clouds.

The structural constraints above are consistent with those in Section 3.3.1. Besides, we further allow AND fanin > 2 so that a tree of c chained AND2s can be written as a single $(c+1)$-input AND node. This makes it trivial both to extract $\mathscr{F}$ from a given abstract XAG and to instantiate an abstract XAG realizing a target $\mathscr{F}$.

Figure 6.4 illustrates the idea on the running example ($^{\#}2888a000$): the abstract XAG in Figure 6.4 a has two AND3s arranged in two levels, hence its AND fence is $\mathscr{F} = \{2,2\}$ (each level represents a tree of two AND2s), shown in Figure 6.4 b.

6.5.3 Exact Synthesis of Ciphertext–Optimal X1Gs

Problem Formulation

We solve a sequence of SAT instances. For a candidate AND fence $\mathscr{F}$, we build a *skeletal* abstract XAG whose AND nodes (with specified arities and levels) match $\mathscr{F}$, but whose XOR-cloud fanins are unknown, like the one shown in Figure 6.4 b. A SAT encoding (details omitted as they are similar to the ones introduced in Section 3.3.2) assigns those fanins so that the abstract XAG realizes the target Boolean function f.

To guarantee optimality, we enumerate fences in non-decreasing ciphertext cost (cf. Eq. (6.6)). A solution found for the first satisfiable fence is thus *ciphertext–optimal* after mapping (Algorithm 6.1).

Algorithm 6.2: Agile exact synthesis with AND fences

Input: Boolean function f; library of fences Lib (sorted by Eq. (6.6))
Output: Ciphertext–optimal abstract XAG for f, or `NULL`

```
fmc ← functional MC of f
foreach 𝓕 ∈ Lib do
    if smc(𝓕) < fmc then
        continue
    N ← SAT(AbstractXAG(f, 𝓕))
    if N ≠ NULL then
        return N
return NULL
```

We pre-compute the set of candidate fences: for structural MC $s \in \{1,\ldots,6\}$ (the known upper bound on functional MC for six-variable functions [39]), all ordered positive-integer partitions of s are included. The order matters because different levels yield different topologies. This results in 63 fences. We further prune using f's functional MC $fmc(f)$: fences with $smc(\mathscr{F}) < fmc(f)$ can be skipped.

Applied to $^{\#}2888a000$, Algorithm 6.2 finds the abstract XAG in Figure 6.4 a, which maps (Algorithm 6.1) to the ciphertext–optimal X1G of Figure 6.3 b.

On XOR Counts and Decomposition

Abstract XAGs hide XOR structure inside clouds. When converting to an implementable XAG, each cloud must be decomposed into XOR2s. A naive decomposition may introduce redundant (functionally equivalent) XOR2s; even optimal decompositions will often exceed the minimum XOR2 count because the exact formulation does not constrain XOR fanins. We *intentionally* leave XOR unconstrained:

(i) with free-XOR, evaluation cost is insensitive to XOR2 count; and

(ii) relaxing XOR greatly enlarges the solution space, accelerating SAT search.

After decomposition and Algorithm 6.1, we still obtain ciphertext–optimal X1Gs, thereby overcoming the limitation discussed in Section 6.4.2.

6.5.4 Database Generation (Up-To-Six-Variable Functions)

While agile, exact synthesis remains exponential. We therefore use it offline to build a functionally complete database of optimal X1Gs for all up-to-six-input functions. We reduce the synthesis effort through *spectral classification* (cf. the "Boolean Classification Techniques" subsubsection in Section 2.1.2): only 150 357 representative functions remain, each synthesized once.

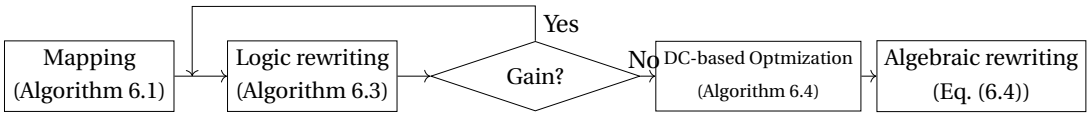


Figure 6.5: Proposed flow for X1G optimization: (i) seed an X1G via the algebraic mapping of Algorithm 6.1; (ii) iteratively improve it using the database-driven logic rewriting of Algorithm 6.3; (iii) post-process with SAT-based SDC elimination (Algorithm 6.4) and constant-driven algebraic rewriting (Eq. (6.4)).

Theorem 6.5.1. *Let f and g be spectrally equivalent, and let f_1 be a ciphertext–optimal X1G for f. Applying the corresponding sequence of spectral operations to f_1 yields an X1G g_1 that is ciphertext–optimal for g.*

Proof. If not, there exists g_2 for g with fewer ONEHOTs than g_1. Applying the inverse spectral operations to g_2 produces an implementation of f with fewer ONEHOTs than f_1, contradicting the optimality of f_1. □

Representatives follow [124]. For each representative, we run Algorithm 6.2 to synthesize an abstract XAG implementation, decompose XOR clouds to obtain an XAG, and map it to an X1G via applying Algorithm 6.1. The resulting database of 150357 ciphertext–optimal X1Gs powers the database-driven rewriting stage of our large-scale X1G optimization flow (next section).

6.6 A Logic Optimization Flow for X1Gs

This section presents a practical three–stage flow to synthesize high-quality X1G implementations for large Boolean functions. The flow takes as input XAGs pre-optimized for low structural MC using prior art [171, 118], seeds an initial X1G by algebraic mapping, and then iteratively improves it using a database-driven rewriting engine, followed by lightweight post-processing.

Stage One: seeding by algebraic mapping. Given an XAG N_{xag} optimized for low structural MC, we first produce a functionally equivalent X1G N_{x1g} by applying the algebraic mapping in Algorithm 6.1: pairs of adjacent AND2s are replaced using Eq. (6.2), and remaining single AND2s via Eq. (6.3). This step is linear-time and provides a strong baseline but, as discussed in Section 6.4.2, is not guaranteed to be garbling-cost optimal.

Stage Two: database-driven logic rewriting. Starting from N_{x1g}, we repeatedly apply a peephole rewrite based on k-feasible cuts and a precomputed database of ciphertext-optimal X1Gs for small functions (the spectral representatives from Section 6.5). For each cut, we extract its local function, look up (via spectral canonicalization) the optimal X1G implementation and cost, and commit the replacement that maximally reduces #ONEHOTs. We iterate until a full pass yields no improvement (Algorithm 6.3).

Algorithm 6.3: Database-driven X1G rewriting (one pass)

Input: X1G network N; database `db` of optimal X1Gs for small functions
Output: Updated N (non-increasing #OneHots)

```
// Recompute cuts only after a successful rewrite
cuts ← ENUMERATE_CUTS(N)
changed ← false
foreach node n in N (topological order) do
    foreach k-feasible cut C ∈ cuts[n] do
        extract local cone G_C and its function f_C
        (r, o) ← SPECTRAL_CANONICALIZE(f_C)
        (G_r*, cost*) ← db[r]                    // optimal impl. and #OneHots
        G_C* ← APPLY_OPS(G_r*, o)                // map back to f_C
        record gain Δ_C ← #ONEHOT(G_C) − #ONEHOT(G_C*)
    choose C' = arg max_C Δ_C
    if Δ_C' > 0 then
        replace G_C' by G_C'* in N
        N ← CLEANUP(N)
        cuts ← ENUMERATE_CUTS(N)
        changed ← true
return N                  // stops when a pass returns with changed = false
```

Stage Three: post-processing. Finally, we reduce the remaining OneHots using two inexpensive passes:

(i) an SDC-based transformation that replaces a OneHot by XOR3 when input pattern $(1,1,1)$ is unreachable (Section 6.6.2), and

(ii) constant-propagation followed by Eq. (6.4) (any ONEHOT with a constant 0 input becomes XOR2).

These passes do not change topology (other than local node replacement) and thus are safe late in the flow.

6.6.1 Database-Driven Logic Rewriting

We adopt a standard cut-based, greedy rewriting framework (Algorithm 6.3) specialized to the X1G cost model, where the objective is the total number of ONEHOT nodes. For scalability, we use dynamic k-feasible cut enumeration [63] and spectral canonicalization to index the database (Section 6.5); this preserves optimality within a cut under the cost metric and exploits Theorem 6.5.1.

Discussion.

(i) *Correctness.* By construction, each committed replacement is functionally equivalent

Algorithm 6.4: SDC-based OneHot→XOR reduction

Input: X1G network N
Output: Updated N

$P \leftarrow$ TSEITIN_ENCODE(N) `// CNF of whole network`
foreach ONEHOT *node n with fanins* $\{x_1, x_2, x_3\}$ **do**
  let (ℓ_1, ℓ_2, ℓ_3) be the CNF literals for (x_1, x_2, x_3)
  if SAT$(P \wedge \ell_1 \wedge \ell_2 \wedge \ell_3) =$ *UNSAT* **then**
    replace n by XOR3(x_1, x_2, x_3)
return N

and non-increasing in #ONEHOTs.

(ii) *Complexity.* Each pass is quasi-linear in the number of cuts; spectral canonicalization and database lookup are $\mathcal{O}(1)$ on average for the small arities used.

6.6.2 Don't-Care–Based OneHot Reduction

A key observation is that ONEHOT and XOR3 differ only on input pattern $(1,1,1)$:

$$\text{ONEHOT} :^{\#} \underline{0}0010110, \qquad \text{XOR3} :^{\#} \underline{1}0010110.$$

Therefore, if a OneHot node n *never* receives $(1,1,1)$ at runtime (i.e., this local pattern is an SDC), we can replace it by XOR3 at zero garbling cost. We detect this with a SAT query formed by Tseitin encoding [173], as summarized by Algorithm 6.4.

6.6.3 Constant-Driven Algebraic Post-Processing

Lastly, we run constant propagation and apply the identity revealed by Eq. (6.4), namely,

$$\text{ONEHOT}(0, x_1, x_2) = \text{XOR2}(x_1, x_2),$$

to remove any residual ONEHOT whose one input is constant 0. This strictly reduces #ONEHOTs and never increases depth. Meanwhile, we *do not* use Eq. (6.3) in this stage because it would reintroduce AND2 nodes and leave the X1G domain.

Putting it together. The overall flow in Figure 6.5 repeatedly alternates Stage two passes until convergence, then applies Stage three once. In our experiments, this converges quickly (few passes) and consistently improves over the algebraic seed, closing the gap identified in Section 6.4.2 while retaining scalability.

Exact Synthesis Engine	#Solutions	Accum. #OneHots	Accum. Time [s]
Baseline [164, 193]	43	113	1354.58
Ours (Algorithm 6.2)	48	120†	40.23

† Excluding the five instances where the baseline produced no solution, our aggregate #OneHots decreases to 105.

Table 6.2: Synthesizing optimal X1Gs for the 48 representatives of five-variable spectral-equivalence classes.

6.7 Experimental Results

This section evaluates the two technical contributions introduced in this chapter:

1. the exact-synthesis formulation in Section 6.5 for agilely producing ciphertext–optimal X1Gs for small functions, and
2. the three–stage optimization flow in Section 6.6 for scalable improvement of X1Gs for practical designs.

All experiments were run on a machine equipped with an Apple M1 Max and 32GB RAM.

6.7.1 Evaluation of the Exact-Synthesis Formulation

We compare our approach (Algorithm 6.2) against a state-of-the-art exact engine (*baseline*) following the *single–selection–variable* (SSV), area-oriented SAT encoding [164], augmented with the heuristic guidance from functional MC [193]. Both solvers are implemented using the C++ reasoning library `bill`[4] with `Glucose`[5] as the underlying SAT solver; each SAT instance is capped at 100 000 conflicts.

The benchmark set comprises the 48 representatives of the spectral-equivalence classes for up-to-five-variable Boolean functions. Results are summarized in Table 6.2. The baseline fails to find *any* solution for five functions and returns suboptimal solutions (more OneHots) for seven others, while our formulation synthesizes solutions for all 48 and finds strictly better implementations in those ambiguous cases. In aggregate, our method is 33.67× faster on average (40.23s vs. 1354.58s) while delivering strictly fewer or equal OneHots per instance.

Why the improvement? Our formulation explicitly embraces the GC cost model: since XORs are free under Free-XOR, we *remove* constraints on XOR counts to inflate the satisfying space while steering the search with AND fences, which determine #OneHots after mapping. This substantially improves SAT solvability without sacrificing optimality under the garbling-cost (i.e., #ciphertexts) objective.

[4] https://github.com/lsils/bill
[5] https://www.labri.fr/perso/lsimon/research/glucose/

6.7.2 Evaluation of the X1G Optimization Flow

We assess the full flow on three widely used benchmark suites:

(i) EPFL combinational designs [5] (reported separately for arithmetic vs. random/control circuits; cf. Section 2.3.1),

(ii) standard cryptographic kernels (cf. Section 2.3.2), and

(iii) MPC application circuits [152] (cf. Section 2.3.3).

For each design, the starting point is the best available XAG from the literature [152, 171, 118]. GC cost is calculated as $2 \cdot \#\text{AND2}s$. We then run our flow (Figure 6.5) to obtain X1G implementations, where an X1G's cost is $2 \cdot \#\text{ONEHOT}s$ (cf. Table 6.1).

Implementation details. We use `mockturtle` [165] for networks and cuts; spectral canonicalization limits the number of attempted operations to 100 000 per cut (if unmatched, the cut is skipped); SDC checks (Algorithm 6.4) use `percy`[6] with `MiniSAT` [78], also capped at 100 000 conflicts per query. For very large designs (> 30k nodes), we bypass the SDC pass to keep runtime practical.

EPFL results. Table 6.3 reports per-stage costs (#ciphertexts), total reduction, and runtime. We observe:

(i) the XAG→X1G-mapping alone already reduces cost by 2.41% (arithmetic) and 11.90% (random/control), confirming the intrinsic ciphertext efficiency of ONEHOT as a nonlinearity provider;

(ii) the database-driven logic rewrite contributes the bulk of savings (on average, 62.77% of total reduction at 84.23% of runtime), while

(iii) the post-processing stages are lightweight yet essential to capture residual opportunities missed by the greedy rewrite.

Overall geometric-mean reductions are 7.34% (arithmetic) and 26.14% (random/control).

Case analyses. For the EPFL benchmark suite, `adder`, `barrel shifter`, and `decoder`, no reduction occurs. These designs contain AND2s that are already *isolated*; after mapping, the resulting ONEHOTs satisfy Eq. (6.3) (namely, with a constant 1 input), which offers no ciphertext advantage. Conversely, `log2`, `sine`, and `voter` exhibit substantial adjacent-AND2 structure; the flow exploits Eq. (6.2) and database-guided merges to unlock more substantial savings.

[6] Availableat:https://github.com/lsils/percy

Benchmark	SOTA	X1G Optimization Flow (#ciphertexts)				Red.	Time [s]
		Mapping	Logic rewrite	SDC	Alg. rewrite		
Arithmetic							
adder	256	256	256 (1†)	256	256	0.00%	1.72
barrel shifter	1664	1664	1664 (1)	1664	1664	0.00%	1.88
divider	10 264	9882	9294 (6)	9288	9288	9.51%	129.31
log2	17 546	16 986	15 584 (4)	15 318	15 314	12.72%	532.73
max	1744	1662	1636 (2)	1636	1636	6.19%	6.71
multiplier	15 170	15 022	14 698 (3)	14 694	14 694	3.14%	116.10
sine	3918	3724	3266 (5)	3244	3242	17.25%	214.06
square-root	10 434	10 176	9406 (5)	9400	9400	9.91%	288.12
square	9192	9052	8662 (4)	8648	8648	5.92%	58.63
GEOMEAN	**4 503.95**	**4 395.44**	**4 186.17**	**4 173.57**	**4 173.16**	**7.34%**	
Random/Control							
round-robin arbiter	2 348	2 260	1 488 (2)	1 488	1 488	36.63%	62.23
coding-cavlc	788	656	524 (2)	512	512	35.03%	39.57
ALU control unit	90	82	74 (3)	74	74	17.78%	2.87
decoder	656	656	656 (1)	656	656	0.00%	1.93
i2c controller	1 114	1 004	886 (3)	872	872	21.72%	34.75
int to float converter	170	144	132 (3)	130	130	23.53%	9.15
memory controller	9 390	8 298	7 264 (5)	7 156	7 156	23.79%	296.11
priority encoder	646	592	450 (2)	450	450	30.34%	16.06
look-ahead XY router	186	126	114 (2)	114	114	38.71%	7.29
voter	8 514	7 848	6 344 (6)	6 242	6 242	26.69%	244.20
GEOMEAN	**850.80**	**749.59**	**633.83**	**628.75**	**628.43**	**26.14%**	

† Parentheses in the "Logic rewrite" column show the number of global passes until saturation.

Table 6.3: Results on EPFL benchmark suite: garbling cost (#ciphertexts) per stage and total reduction.

Stage-wise contributions and runtime. Across EPFL benchmarks, the algebraic mapping and the logic rewriting contribute 62.77% and 32.64% of the overall reduction, respectively, consuming 0.07% and 84.23% of runtime. SDC and constant-driven algebraic rewriting are quick but important for closing residual gaps. The most time-consuming instances are large arithmetic designs (`log2`, `sqrt`, `mem_ctrl`) due to many profitable cuts and the breadth of the database search.

Cryptographic and MPC results. Table 6.4 reports analogous data. For cryptographic circuits, the geometric-mean reduction is 13.51%; for MPC applications, 4.34%. The smaller gains on MPC are consistent with two observations:

(i) ANDs are intentionally spaced (few adjacent AND2 pairs), and

(ii) these circuits are already designed to be GC-friendly, leaving less room for optimization.

Takeaways. Based on the results reported in Tables 6.3 and 6.4, we summarize that:

(i) ONEHOT is a more ciphertext-efficient nonlinearity provider than AND2 in many practical circuits;

(ii) a pure algebraic map already helps, but database-driven rewriting is key to consistent improvements;

(iii) the benefit is application-dependent—benchmarks engineered to be GC-friendly (namely, the MPC circuits) leave fewer optimization opportunities, whereas heterogeneous control logic (random/control circuits from the EPFL benchmark suite) benefits most from our flow.

6.8 Discussion

6.8.1 Summary of Contributions

This chapter challenged the XAG-centric view for low-cost GC generation and introduced X1G as a ciphertext-efficient alternative. Concretely, we:

1. quantified nonlinearity efficiency via *MC compactness* and identified ONEHOT as a superior primitive;
2. provided an algebraic XAG→X1G mapping with provable per-instance savings;
3. developed an *agile* exact-synthesis formulation using AND fences and abstract XAGs that scales to all up-to-six-input spectral representatives; and
4. assembled a practical, three-stage X1G optimization flow that consistently reduces garbling cost across diverse benchmark suites.Geometric-mean reductions of 7.34% (EPFL arithmetic), 26.14% (EPFL random/control), 13.51% (cryptographic), and 4.34% (MPC) were achieved with reasonable runtime.

6.8.2 Optimization Effort vs. Runtime

Our experiments highlight a familiar tension: higher-effort logic optimization generally yields lower garbling cost but at nontrivial runtime. This suggests tailoring the flow to the deployment model. For *on-the-fly* circuit generation, where synthesis latency can bottleneck end-to-end MPC, lightweight passes are preferable: the algebraic mapping (Algorithm 6.1) and the constant-driven clean-up via Eq. (6.4) together deliver noticeable savings at essentially negligible cost. Conversely, for *reusable* benchmarks (e.g., typical MPC tasks and cryptographic primitives), investing in high-effort optimization is justified: database-driven rewriting (Algorithm 6.3) and SDC-based reduction (Algorithm 6.4) systematically harvest additional opportunities. Importantly, publishing optimized netlists does *not* weaken security — the privacy of GC stems from the garbling process and encrypted tables, not from obscurity of the underlying Boolean network.

6.8.3 Where X1G Helps and Where It Does Not

The mapping and rewriting results confirm that ONEHOT is a more ciphertext-efficient nonlinearity provider than AND2 in many practical designs, especially those with adjacent-AND2 structure that can be folded using Eq. (6.2). At the same time, benchmarks engineered to be GC-friendly (e.g., many MPC circuits) already space nonlinearities, leaving limited opportunities for optimization. These observations motivate *structure-aware* flows that detect when X1G is advantageous and otherwise avoid unproductive transformations.

6.8.4 Future Work

Several directions naturally follow:

- *Scenario-aware flow selection.* Build a portfolio of X1G flows (lightweight vs. heavy) and a learned selector that predicts, from quick structural features (e.g., density of adjacent AND2, fanout patterns), which flow maximizes garbling-cost reduction per unit time.
- *Heterogeneous representations.* While this chapter advocates X1G, some subcircuits may be better served by alternative primitives under certain garbling schemes. A heterogeneous mapper that mixes X1G with carefully chosen non-ONEHOT primitives could capture such cases while preserving free-XOR compatibility.
- *Richer database and matching.* Extending the exact-synthesis database beyond six inputs via improved SAT encodings and tighter pruning, plus faster spectral canonicalization, would broaden the reach of database-driven rewriting.
- *Sharper DC usage.* Beyond SDCs, incorporating scalable approximations to ODCs (e.g., windowed SAT or implication-based filters) may enable additional ONEHOT→XOR3 substitutions (Eq. (6.4)) with modest overhead.
- *Tighter coupling with scheme-level advances.* Our logic-level methods assume free-XOR and standard point-and-permute; future garbling schemes that improve symmetric gates or support new gadgets can be surfaced as cost models inside the same optimization framework, allowing the representation to co-evolve with protocol advances.

6.8.5 A Question Pointing Forward

While we now have an automated way to produce low-cost garbled circuits for a given function, must the *entire* function be garbled? If portions of the computation depend only on a single party's data, can we maximize such *local evaluation* and minimize the GC core accordingly? Pursuing this orthogonal and complementary direction opens a new avenue for reducing garbling cost and overall communication — an avenue we explore in the next chapter.

Benchmark	SOTA	X1G Optimization Flow (#ciphertexts)				Red.	Time [s]
		Mapping	Logic rewrite	SDC	Alg. rewrite		
Cryptographic							
AES (KeyExp)	10 880	10 240	10 240 (2)	10 240	10 240	5.88%	41.17
AES (No KeyExp)	13 600	12 800	12 800 (2)	12 800[†]	12 800	5.88%	50.22
DES (KeyExp)	13 830	13 346	12 864 (4)	12 482	12 474	9.80%	533.58
DES (No KeyExp)	13 666	13 194	12 690 (5)	12 280	12 278	10.16%	525.35
Comp. 32-bit SLT	168	138	120 (3)	120	120	28.57%	6.81
Comp. 32-bit SLTEQ	174	152	134 (2)	134	134	22.99%	7.24
Comp. 32-bit ULT	168	138	120 (3)	120	120	28.57%	6.81
Comp. 32-bit ULTEQ	174	152	134 (2)	134	134	22.99%	7.26
MD5	18 734	18 734	18 734 (1)	18 734	18 732	0.01%	27.61
SHA-1	22 966	22 834	22 636 (3)	22 636	22 636	1.44%	107.32
SHA-256	52 928	51 832	50 086 (3)	50 086	50 086	5.37%	335.79
GEOMEAN	**3 322.25**	**3 066.13**	**2 890.11**	**2 873.61**	**2 873.38**	**13.51%**	
MPC							
auction_2_16	194	194	194 (1)	194	194	0.00%	1.92
auction_2_32	386	386	386 (1)	386	386	0.00%	1.92
auction_3_16	464	464	460 (2)	460	460	0.86%	3.04
auction_3_32	912	912	908 (2)	908	908	0.44%	3.11
auction_4_16	990	990	986 (2)	986	986	0.40%	3.26
auction_4_32	1 950	1 950	1 946 (2)	1 946	1 946	0.21%	3.44
knn_comb_1_8	1 108	1 108	1 100 (2)	1 100	1 100	0.72%	4.42
knn_comb_1_16	2 324	2 324	2 300 (2)	2 300	2 300	1.03%	4.87
knn_comb_2_8	1 762	1 726	1 676 (4)	1 676	1 676	4.88%	10.40
knn_comb_2_16	3 838	3 754	3 648 (4)	3 648	3 648	4.95%	15.21
knn_comb_3_8	2 120	2 108	2 082 (3)	2 082	2 082	1.79%	7.90
knn_comb_3_16	4 788	4 760	4 694 (3)	4 694	4 694	1.96%	13.64
voting_1_3	14	14	12 (2)	12	12	14.29%	1.92
voting_1_4	30	28	26 (2)	26	26	13.33%	2.18
voting_2_2	42	40	38 (2)	38	38	9.52%	2.93
voting_2_3	110	110	110 (1)	110	110	0.00%	2.37
voting_2_4	208	208	204 (2)	204	204	1.92%	3.72
voting_3_4	550	550	536 (2)	536	536	2.55%	7.47
stable_matching_4_8	32 002	29 416	27 824 (6)	27 824	27 824	13.06%	379.88
stable_matching_8_8	119 546	108 320	105 086 (3)	105 086	105 086	12.10%	502.04
GEOMEAN	**790.95**	**777.04**	**757.74**	**756.64**	**756.64**	**4.34%**	

[†] Gray entries indicate the SDC stage was skipped due to size (>30k nodes).

Table 6.4: Results on Cryptographic & MPCircuit benchmark suites: garbling cost (#ciphertexts) per stage and total reduction.

7 Redefining Cost in Garbled Circuits: Joint Multiplicative Complexity

7.1 Motivation

Garbled circuits (GC) offer a general-purpose path to *secure multi-party computation* (SMPC) with constant-round protocols and a clean Boolean abstraction. Yet the price of this generality is well known: the secure evaluation of Boolean gates must be realized through ciphertext exchange (i.e., garbled tables), and thus dominates both communication and computation cost.

However, it is rarely highlighted that this applies only to those gates whose outputs depend on inputs held by different parties. In contrast, any operation whose value depends on a single party's inputs can be performed *locally*, with no interaction and negligible overhead. This simple observation raises a practical question that most GC pipelines leave implicit: *Given a target function f, how much of its Boolean implementation truly requires joint evaluation, and how much can (and should) be pushed into local, non-interactive computation?*

Recent systems hint at the power of this perspective in domain-specific settings. For secure analytics, `Senate` decomposes SQL operators and handcrafts circuit templates that minimize joint work across `filtering`, `joins`, and set operations [145]. In secure model adaptation, `Marill` reduces joint cost by *freezing* layers that need no retraining, thereby avoiding interaction over large swaths of the network [148]. Similar ideas appear in data-processing platforms that distinguish local preprocessing from secure aggregation [178]. These systems demonstrate a common principle — *think locally before compute jointly* — but they rely on manual, application-specific engineering and do not generalize to arbitrary Boolean computations.

This chapter pursues that principle in a protocol-agnostic, compiler-integrated way. Our goal is to operate directly on gate-level circuits (the common currency of GC toolchains) and systematically *maximize local computation* while preserving functionality and privacy. Concretely, we want an automated flow that, given a Boolean circuit for $f : \{0,1\}^n \mapsto \{0,1\}^m$ with inputs partitioned across parties,

(i) identifies subcircuits whose support lies entirely within one party,

(ii) restructures the network so that these subcircuits are evaluated locally, and

(iii) minimizes the residual "joint frontier" that must be garbled.

Why is this worth doing on top of the logic-level savings developed in the previous chapter? Representation choices (e.g., X1G vs. XAG) reduce the *cost per joint gate*; maximizing locality reduces the *number of joint gates*. These two axes are complementary. Even after aggressive garbling-cost optimization, joint gates remain the primary driver of communication; shrinking their population directly cuts ciphertext volume. In this chapter, we deliberately adopt the more classical XAG model. This is out of the understanding that, our goal here is to demonstrate a direct conceptual progression from prior work: shifting the optimization objective from *multiplicative complexity* (MC) to the refined metric of *joint multiplicative complexity* (JMC); working within the standard XAG framework makes this transition transparent, allowing the contribution of the new cost model to stand on its own. One shall note that the methodology proposed in this chapter can be seamlessly integrated with improved representations such as X1G, thereby further amplifying the benefits.

Designing a general, automated solution is non-trivial. Several challenges arise:

- **Discovery.** Determining which gates are truly single-party requires precise *support analysis* under fanout, reconvergence, and sharing.
- **Restructuring.** Pushing local computation toward the inputs must not increase the downstream joint region disproportionately (e.g., via duplication) or break logic sharing opportunities.
- **Cost awareness.** The transformation must weigh reduced joint work against any local expansion, under realistic GC cost models.

Guided by these considerations, and inspired by the application-specific gains reported in prior systems [145, 148, 178], we develop a gate-level framework that automatically restructures circuits to *minimize joint computation* before garbling. As we will show, this simple shift in granularity yields substantial savings across diverse benchmarks.

7.2 Preliminaries

This chapter assumes the GC protocol and circuit notation introduced in Chapter 2. We review two ingredients that we will use repeatedly:

(i) Boolean decomposition techniques that expose structure and locality in gate-level networks, and

(ii) related work on optimizing garbled circuits — both in terms of per-gate costs and, closer to our focus here, the *amount* of joint computation.

7.2.1 Boolean Decomposition

Boolean decomposition expresses a Boolean function as a composition of simpler sub-functions whose recombination reconstructs the original. For a single-output function $f(\vec{x}) : \{0,1\}^n \to \{0,1\}$ with support $\vec{x} = (x_1, \ldots, x_n)$, a decomposition partitions the variables into k pairwise-disjoint subsets $\vec{x}_1, \ldots, \vec{x}_k$ such that

$$\vec{x} = \vec{x}_1 \cup \cdots \cup \vec{x}_k, \qquad \vec{x}_i \cap \vec{x}_j = \emptyset \text{ for } i \neq j.$$

A valid decomposition then seeks sub-functions $\vec{g}_1(\vec{x}_1), \ldots, \vec{g}_k(\vec{x}_k)$ and a top-level combiner l satisfying

$$f(\vec{x}) = l\big(\vec{g}_1(\vec{x}_1),\ \vec{g}_2(\vec{x}_2),\ \ldots,\ \vec{g}_k(\vec{x}_k)\big).$$

This divide-and-conquer view is valuable both theoretically and practically: smaller components admit stronger optimization (e.g., SAT-based exact synthesis on tractable subproblems [91]); their optimized implementations can then be combined to yield high-quality networks that are difficult to obtain monolithically.

Shannon Decomposition

A classical, variable-oriented technique expands on a single input x_i and expresses f via its cofactors:

$$\begin{aligned} f(\vec{x}) &= x_i \cdot f(x_1, \ldots, x_i{=}1, \ldots, x_n) + \neg x_i \cdot f(x_1, \ldots, x_i{=}0, \ldots, x_n) \\ &= (x_i \cdot f_{x_i}) \oplus (\neg x_i \cdot f_{\neg x_i}), \end{aligned}$$

where f_{x_i} and $f_{\neg x_i}$ are the positive and negative cofactors [160]. The XOR form is particularly attractive in secure computation because it foregrounds linear combinations that can be handled with free-XOR, pushing non-linearity into the cofactors. Recursing on carefully chosen variables exposes regular structure and supports systematic exploration of alternatives (e.g., in classifier circuits [65]).

Ashenhurst-Curtis Decomposition

Ashenhurst-Curtis decomposition (ACD) is a more general technique [10, 67]. It systematically partitions the input space of a Boolean function into independent components, enabling functional simplification through intermediate abstractions. For a single-output Boolean

function $f(\vec{x})$, the ACD expresses the function in the form

$$f(\vec{x}) = l\big(\vec{g}(\vec{x}_{\mathrm{BS}},\ \vec{x}_{\mathrm{SS}}),\ \vec{x}_{\mathrm{SS}},\ \vec{x}_{\mathrm{FS}}\big),$$

where the input variable set $\vec{x}$ is partitioned into three pair disjoint subsets: $\vec{x}_{\mathrm{BS}}$, $\vec{x}_{\mathrm{SS}}$, and $\vec{x}_{\mathrm{FS}}$, known respectively as the *bound set* (BS), *shared set* (SS), and *free set* (FS). The sub-functions $\vec{g}$ are referred to as the BS functions, as they capture the behavior of f over the bound variables, while the function l serves as the combining logic.

Historically, ACD underpins many LUT-oriented flows for FPGAs [125, 170]. In our setting, variable partitions are also a natural vehicle to *separate per-party support*: when inputs are annotated by ownership, a decomposition that isolates supports within a single party directly surfaces subcircuits eligible for local gate evaluation.

7.2.2 Related work

Optimal garbled-circuit generation

A large body of prior work automates low-cost GC generation by treating it as a problem of *multiplicative complexity (MC) reduction.* The standard recipe is to implement the target function as an XAG and minimize the number of AND nodes (non-linear operations), since XORs are essentially free under free-XOR while ANDs dominate cost [166]. This has led to a rich ecosystem of MC-oriented optimizations and toolflows [172, 171, 118].

Our perspective is complementary. Beyond reducing the *cost per non-linear gate*, we refine the objective to the *amount of non-linear work that truly requires garbling.* Concretely, we will use a notion of *joint multiplicative complexity* (JMC): the number of non-linear gates whose fanin support spans more than one party and thus must be evaluated jointly. By explicitly targeting JMC, we can restructure circuits to maximize local (single-party) computation before applying any scheme-level garbling optimizations.

Joint computation minimization

Several systems reduce joint computation in application-tailored ways. For secure analytics, `Senate` designs operator-specific circuits (`filters`, `joins`, `set intersections`) that minimize communication cost when composed into SQL queries [145]. In secure ML, `Marill` reduces garbling cost during LLM fine-tuning via layer freezing, avoiding joint work where accuracy permits [148]. Data-processing platforms similarly separate local preprocessing from secure aggregation [178].

These approaches demonstrate substantial gains but rely on manual, domain-specific engineering. In contrast, our contribution is an *automated, gate-level, general-purpose* framework: given any Boolean circuit with inputs partitioned by party, we algorithmically identify and

extract local subcircuits, minimize the residual joint region, and thereby cut garbling cost without application-specific templates or human-in-the-loop design.

7.3 Overview

This section presents a high-level overview of our proposed optimization framework for garbled circuit optimization via joint computation minimization. We first introduce a refined cost model based on the notion of *node ownership*, which distinguishes between operations that require secure joint evaluation and those that can be executed locally. We then outline the architecture of our optimization framework, highlighting its cut-based peephole optimization strategy. Finally, we explain why decomposition-driven resynthesis — in particular, ACD (cf. the sub-subsection on "Ashenhurst-Curtis decomposition" in Section 7.2.1) — is a scalable backbone for this task.

7.3.1 Node ownership and refined cost model

Limitations of MC as a cost proxy

While MC serves as a coarse indicator of the garbling cost for a Boolean circuit, it does not fully capture the actual cost incurred in MPC using garbled circuits. As also observed in prior work (e.g., [178]), garbled gates whose execution depends solely on data from a single party need not be part of the MPC protocol. Such gates can be evaluated locally and in plaintext, making them effectively free in terms of cryptographic overhead. Consequently, it is not the total number of AND nodes in the XAG representation of the target function that dictates cost, but rather the subset whose evaluation requires joint inputs from multiple parties.

This insight exposes a key limitation of using MC as the sole cost proxy: it overestimates the cryptographic cost by failing to distinguish between local and joint computation. In practice, identifying and excluding locally derivable operations from the garbled circuit offers a near cost-free efficiency improvement. In this work, we take this observation a step further by asking: beyond passively identifying local computation, can we actively restructure the logic to reduce the number of jointly evaluated operations, possibly at the expense of increased local processing?

To formalize this idea, we introduce the notion of node ownership, a mechanism to statically determine whether a node in a logic network requires joint computation or can be evaluated locally. This refined view enables a more accurate cost model that distinguishes the truly costly components of an MPC task from those that can be offloaded without interaction.

Node ownership

To enable finer-grained cost modeling for GC–based MPC, we introduce the notion of node ownership, which classifies each node in a logic network as either *locally derivable* or *jointly derived*. This distinction reflects whether the node's computation depends solely on inputs from a single party or requires data from multiple parties and must therefore be computed securely within the MPC protocol.

Let $G = (V, E)$ be a logic network with PI nodes $I \subseteq V$, and let $\mathscr{P} = \{\mathscr{P}_1, \mathscr{P}_2, \ldots, \mathscr{P}_t\}$ denote the set of t semi-honest parties participating in the computation. Each PI node $v \in I$ is owned by exactly one party, indicated by a mapping:

$$\mathsf{own} : I \mapsto \mathscr{P},$$

which specifies the party $\mathscr{P}_i \in \mathscr{P}$ that privately holds the value of v.

We define the set of locally derivable nodes with respect to party $\mathscr{P}_i$, denoted $\mathsf{LD}_{\mathscr{P}_i} \subseteq V$, recursively as:

$$v \in \mathsf{LD}_{\mathscr{P}_i} \iff \begin{cases} v \in I \text{ and } \mathsf{own}(v) = \mathscr{P}_i, \\ \text{or} \\ \forall u \in \mathrm{fanin}(v),\ u \in \mathsf{LD}_{\mathscr{P}_i}. \end{cases}$$

In other words, a node is locally derivable by party $\mathscr{P}_i$ if it is either: (1) a PI owned by $\mathscr{P}_i$, or (2) a gate whose fanin nodes are all locally derivable by $\mathscr{P}_i$.

The set of jointly derived nodes, denoted $\mathsf{JD} \subseteq V$, is defined as:

$$\mathsf{JD} = \{v \in V \mid \forall i \in \{1, \ldots, t\},\ v \notin \mathsf{LD}_{\mathscr{P}_i}\}.$$

That is, a node belongs to JD if it is not locally derivable with respect to any single party.

We further define the *frontier set*, denoted $\mathsf{FR} \subseteq \mathsf{JD}$, as:

$$\mathsf{FR} = \{v \in \mathsf{JD} \mid \exists u \in \mathrm{fanin}(v) \text{ s.t. } u \in \mathsf{LD}_{\mathscr{P}_i} \text{ for some } i\}.$$

Intuitively, the frontier consists of jointly derived nodes that are immediately preceded by at least one locally derivable node. This concept is of particular interest, since the goal of this work, namely, minimizing joint computation via maximizing local data processing, can be viewed as a process of "pushing" the frontier from the PI side towards the PO side.

Refined cost model

The limitations of using MC alone as the cost proxy for garbled circuits generation can be illustrated by the example in Figure 7.1. We review two XAG implementations for Boolean

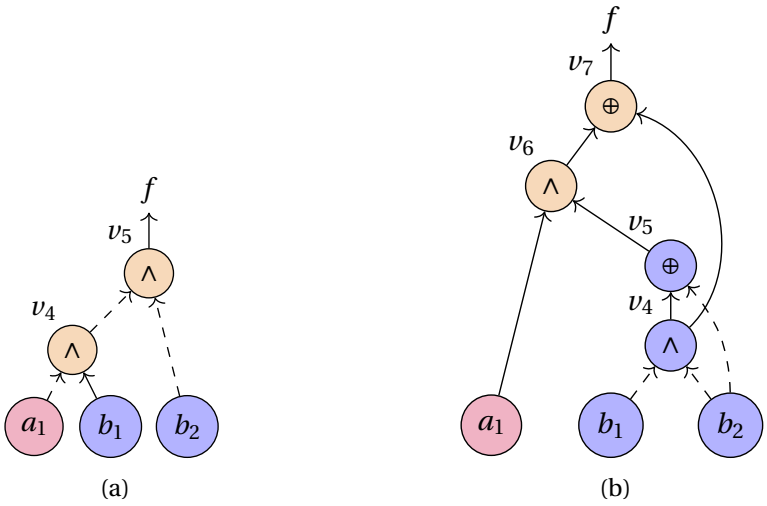


Figure 7.1: Two MC-optimal XAGs for Boolean function f, whose truth table is $^{\#}0b$, under ownership $a_1 \in \mathscr{A}$, $b_1, b_2 \in \mathscr{B}$. Nodes locally derivable by $\mathscr{A}$ (resp. $\mathscr{B}$) are shown in red (resp. blue); jointly derived nodes are orange. (a) Both ANDs are joint JMC = 2. (b) Only one AND is joint JMC = 1.

function f (truth table: `#0b`) under an MPC context: among the three PIs, a_1 originates from party $\mathscr{A}$, while b_1 and b_2 originate from party $\mathscr{B}$.

These two XAGs are both MC-optimal. This follows from the fact that, for this small-scale, three-variable function f, SAT-based exact synthesis [163] can determine that its *functional MC* is 2, meaning that at least two AND gates are required in any XAG implementation. Both implementations meet this lower bound. If MC were used as the sole optimization metric, these two XAGs would be considered equally good; moreover, from a pure graph complexity perspective, the network in Figure 7.1 a might even appear preferable due to having fewer total nodes.

However, incorporating the notion of node ownership reveals a different picture. We annotate nodes locally derivable to $\mathscr{A}$ and $\mathscr{B}$ in red and blue, respectively, and jointly derived nodes in orange. Although both implementations contain exactly two AND gates, in Figure 7.1 a, both AND gates (v_4 and v_5) are jointly derived, each incurring one unit of garbling cost. In contrast, in Figure 7.1 b, only one of the two AND gates (v_6) is jointly derived; the other (v_4) is locally derivable to $\mathscr{B}$ and can be evaluated offline without any MPC overhead. Consequently, when adopting the implementation in Figure 7.1 b for garbled circuit generation, the effective inputs from $\mathscr{B}$ become v_4 and v_5 rather than b_1 and b_2.

This example demonstrates that MC alone cannot identify computation that could be executed outside the GC protocol. Our refined cost model addresses this by focusing on the subset of non-linear nodes (AND nodes) that are jointly derived. Formally, for an XAG $G = (V, E)$, let $\mathsf{JD} \subseteq V$ be the set of jointly derived nodes as defined in the previous sub-subsection. We

Algorithm 7.1: Peephole Optimization Framework

Input: XAG G, synthesis engine Syn, cut size bound κ
Output: Optimized XAG G

1 **foreach** $v \in V(G)$ *in topological order* **do**
2 $\quad \mathscr{C} \leftarrow \mathsf{EnumerateCuts}(G, v, \kappa)$
3 $\quad G_v^{\text{best}} \leftarrow \texttt{NULL}$
4 $\quad jmc_v^{\text{best}} \leftarrow \mathsf{JMC}(G)$
5 $\quad$ **foreach** $c \in \mathscr{C}$ **do**
6 $\quad\quad f_c \leftarrow \mathsf{ExtractFunction}(G, c)$
7 $\quad\quad G_c^{\text{cand}} \leftarrow \mathsf{Syn}(f_c)$
8 $\quad\quad$ **if** $\neg\mathsf{StructurallyDifferent}(G_c^{cand}, c)$ **then**
9 $\quad\quad\quad$ **continue**
10 $\quad\quad$ **if** $\mathsf{JMC}(G_c^{cand}) < \mathsf{JMC}(c)$ **then**
11 $\quad\quad\quad G_{\text{temp}} \leftarrow \mathsf{Replace}(G, c, G_c^{\text{cand}})$
12 $\quad\quad\quad$ **if** $\mathsf{JMC}(G_{temp}) < jmc_v^{best}$ **then**
13 $\quad\quad\quad\quad G_v^{\text{best}} \leftarrow G_{\text{temp}}$
14 $\quad\quad\quad\quad jmc_v^{\text{best}} \leftarrow \mathsf{JMC}(G_{\text{temp}})$
15 $\quad$ **if** $G_v^{best} \neq \texttt{NULL}$ **then**
16 $\quad\quad G \leftarrow G_v^{\text{best}}$
17 **return** G

define the *joint multiplicative complexity* (JMC) as:

$$\mathsf{JMC}(G) = |\{v \in \mathsf{JD} \mid \circ_v = \wedge\}|.$$

The JMC precisely counts the number of AND gates whose evaluation requires joint computation, and thus serves as the target metric for our logic optimization framework. By minimizing JMC rather than MC, we directly reduce the actual garbling cost and improve MPC efficiency.

7.3.2 Logic optimization framework

Our optimization framework adopts a peephole rewriting strategy operating directly at the gate-netlist level of an XAG. The core idea is to iteratively identify small sub-networks, represented by cuts of bounded size, and resynthesize them using a dedicated synthesis engine (introduced in Section 7.4). The objective is to minimize the number of jointly derived AND nodes (i.e., JMC) while preserving functional equivalence. This strategy is inspired by [154], but adapted to our cost model.

As presented in Algorithm 7.1, the process begins with a topological traversal of all nodes in G. For each node v, we enumerate all cuts rooted at v whose number of leaves does not exceed the bound κ (line 2). We initialize placeholders G_v^{best} and jmc_v^{best} (lines 3–4) to store the best candidate implementation for node v.

Each cut c defines a κ'-input, single-output Boolean function f_c, where $\kappa' \leq \kappa$. Once f_c is derived (line 6), the synthesis engine is invoked on the function to obtain a candidate implementation G_c^{cand} (line 7). Candidates structurally identical to the current implementation are skipped.

We perform a two-stage evaluation. In Stage one (line 10), we compare the JMC of the candidate against that of the current cut in isolation. If it improves locally, we run Stage two, a global dry run (line 12): temporarily replace c with G_c^{cand} in G and re-evaluate JMC for the entire network to account for shared sub-structures. If the new JMC beats the best found so far for v, we record it (lines 13-14). Only after all cuts of v are examined do we commit the best recorded implementation (lines 15–16). This ensures that for each root node, we adopt the cut replacement yielding the maximum global cost reduction. This two-stage evaluation strategy enables the framework to pursue aggressive local improvements while maintaining global caution, ensuring that the final implementation minimizes jointly derived computation across the entire network.

7.4 Methodology

Building on the insights from the previous section, we now present the design of our Boolean synthesis engine tailored to minimize jointly derived computation. At the core of our approach is an ACD-based strategy, which enables systematic partitioning of functions into sub-components aligned with party-specific input ownership.

7.4.1 Joint-Compute–Oriented Decomposition

Recall the symbolic formulation of Ashenhurst–Curtis decomposition (ACD) introduced in Section 7.2.1:

$$f(\vec{x}) = l\big(\vec{g}(\vec{x}_{\text{BS}}, \vec{x}_{\text{SS}}), \vec{x}_{\text{SS}}, \vec{x}_{\text{FS}}\big),$$

where a decomposition is successful if one can construct a set of BS functions $\vec{g}$ over the variables $(\vec{x}_{\text{BS}}, \vec{x}_{\text{SS}})$ and a top function l such that together they realize f.

Following recent advances in scalable ACD realization [170], we adopt the standard interpretation of l as a top-level multiplexer (MUX) driven by a set of FS functions $\vec{h}$ defined over $\vec{x}_{\text{FS}}$. In this view, the BS functions $\vec{g}$ and the SS variables jointly serve as the select signals of the MUX, while the FS functions $\vec{h}$ form its data inputs. From the perspective of the MUX structure, BS and SS variables play an indistinguishable role, as both contribute to the select lines, except that SS variables appear both directly and through the BS functions $\vec{g}(\vec{x}_{\text{BS}}, \vec{x}_{\text{SS}})$. This observation allows us to conceptually merge the BS and SS into a single set of "selector variables," which we continue to denote by $\vec{x}_{\text{BS}}$ for simplicity.

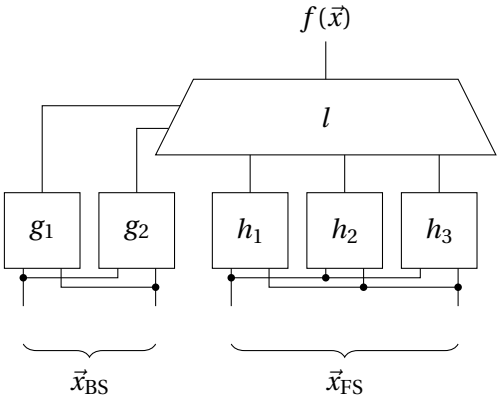


Figure 7.2: Illustration of rephrased Ashenhurst–Curtis decomposition. A four-variable function is decomposed into three free-set functions $\vec{h} = \{h_1, h_2, h_3\}$ and two bound-set functions $\vec{g} = \{g_1, g_2\}$. The top function l is implemented as a multiplexer, using $\vec{g}$ to select among the FS functions $\vec{h}$.

Under this consolidation, the symbolic form of ACD can be written equivalently as:

$$f(\vec{x}) = l\big(\vec{g}(\vec{x}_{\mathrm{BS}}),\ \vec{h}(\vec{x}_{\mathrm{FS}})\big),$$

where l is explicitly realized as the top MUX. This equivalent formulation preserves the full generality of ACD while significantly simplifying the variable-partitioning problem — an advantage that becomes central to designing an efficient JMC-oriented resynthesis engine.

An illustration of this formulation is given in Figure 7.2, which shows the decomposition of a four-variable function into three FS functions $\vec{h} = \{h_1, h_2, h_3\}$ and two BS functions $\vec{g} = \{g_1, g_2\}$.

Guided by this blueprint, our synthesis engine is structured into three stages: (1) partition variables and derive the FS functions ($\vec{h}$); (2) assign encodings to the FS functions and construct the BS functions ($\vec{g}$); (3) synthesize the top MUX (l). The remainder of this subsection describes the first two stages; the construction of the MUX is presented in Section 7.4.2.

Variable Partitioning and Free Set Function Construction

Our variable partitioning procedure is described in Algorithm 7.2. Given a cut c with leaves L and its function, we first separate exclusively owned leaves by party (lines 1-2). This explicitly identifies signals that can be assigned to a single party's local computation, and therefore, on the FS side.

Once the FS set $\vec{x}_{\mathrm{FS}}$ is determined, the corresponding FS functions are uniquely fixed. These functions are obtained by enumerating all value assignments of the BS variables: for each assignment, the restricted function defines one FS function. Collecting the distinct outcomes yields the FS function set $\vec{h}$. This process is most naturally visualized through the truth table

Algorithm 7.2: Ownership-Aware Variable Partitioning

Input: Cut c with leaf set L,
ownership map $\mathsf{own} : L \mapsto \mathscr{P}$
Output: Set of candidate partitions $\mathscr{S}$

foreach $\mathscr{P}_i \in \mathscr{P}$ **do**
  $L_i \leftarrow \{\ell \in L \mid \mathsf{own}(\ell) = \mathscr{P}_i\}$
$\mathscr{S} \leftarrow \emptyset$
foreach $\mathscr{P}_i \in \mathscr{P}$ **do**
  $\vec{x}_{\mathrm{FS}} \leftarrow L_i$
  $\vec{x}_{\mathrm{BS}} \leftarrow (L \setminus L_i)$
  if $|\vec{x}_{FS}| > 1$ **then**
    add $(\vec{x}_{\mathrm{BS}}, \vec{x}_{\mathrm{FS}})$ to $\mathscr{S}$
return $\mathscr{S}$

b_2	b_1	a_1	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

reorder variables →

$\vec{x}_{\mathrm{BS}}$	$\vec{x}_{\mathrm{FS}}$			
a_1	b_2	b_1	f	
0	0	0	1	
0	0	1	0	h_1
0	1	0	0	
0	1	1	0	
1	0	0	1	
1	0	1	1	h_2
1	1	0	0	
1	1	1	0	

Figure 7.3: Deriving FS functions for the partition $\vec{x}_{\mathrm{FS}} = \{b_1, b_2\}$, $\vec{x}_{\mathrm{BS}} = \{a_1\}$. Reordering the truth table by a_1 reveals two distinct FS functions: $h_1 = \neg b_1 \wedge \neg b_2$ and $h_2 = \neg b_2$.

representation.

Running example. In the example shown in Figure 7.1, among the three primary inputs, a_1 is exclusively owned by party $\mathscr{A}$, while b_1 and b_2 are exclusively owned by party $\mathscr{B}$. According to Algorithm 7.2, this leads to a single valid partition: $\vec{x}_{\mathrm{FS}} = \{b_1, b_2\}$ and $\vec{x}_{\mathrm{BS}} = \{a_1\}$. As shown in Figure 7.3, reordering the variables in the truth table reveals two distinct FS functions, namely $\vec{h} = \{h_1, h_2\}$, with truth tables `#1` and `#3`. These correspond to the Boolean expressions $h_1 = \neg b_1 \neg b_2$ and $h_2 = \neg b_2$.

Encoding and Bounded-Set Function Derivation

The number of FS functions $|\vec{h}|$ directly determines the number of BS functions required to distinguish them. In principle, $\log_2 |\vec{h}| \leq |\vec{g}| \leq |\vec{h}|$. However, in an MPC context, the computation of BS functions typically involves multiple parties, thereby contributing to the jointly

derived computation. To reduce this cost, we enforce the constraint

$$|\vec{g}| = \lceil \log_2 |\vec{h}| \rceil,$$

ensuring that the number of BS functions is minimized while still preserving the ability to encode all FS functions uniquely.

This formulation naturally connects the problem to an *encoding* task: each FS function must be assigned at least one $\lceil \log_2 |\vec{h}| \rceil$-bit code that specifies its selection by the BS functions at the input of the top multiplexer l. In the simple case previously introduced in Figure 7.3, two FS functions are present, i.e., $|\vec{h}| = 2$. The encoding is straightforward: one function is assigned code 0 and the other code 1. This implies a single BS function g_1, implemented directly over the BS variable a_1. Depending on the assignment, g_1 may realize either the identity $g_1 = a_1$ or the negation $g_1 = \neg a_1$, both of which yield equivalent garbling costs.

While the encoding is trivial for small $|\vec{h}|$, the general case involves a significantly larger design space. Each FS function must be mapped to a codeword such that all functions remain distinguishable, and the induced BS functions are minimized in both number and support size. This can be interpreted as a *constrained encoding* problem and solved via a two-step *encoding-minimization* process [170]:

1. **Candidate code generation**: Each FS function is associated with at least one codeword of length $\lceil \log_2 |\vec{h}| \rceil$. Codewords are selected so that different FS functions are mapped to distinct encodings, guaranteeing that they can be separated by the BS signals.
2. **Support minimization**: For each bit position of the codewords, the induced Boolean function over the BS variables defines a candidate BS function. Since multiple encodings may be valid for a given FS function, the assignment is optimized so that the resulting BS functions have minimal support, i.e., they depend on as few BS variables as possible. This reduces the complexity of the BS side of the decomposition, thereby limiting the amount of joint computation required.

The outcome of this procedure is a strict encoding, where each FS function is assigned a unique code, leading to a decomposition that balances distinguishability with minimal joint computation. The codes correspond to a compact set of BS functions $\vec{g}$.

Running example. With $|\vec{h}| = 2$, a single BS bit suffices. Assigning $h_1 \mapsto 0$, $h_2 \mapsto 1$ yields $g_1 = a_1$ (or equivalently $\neg a_1$ if codes are swapped). Figure 7.4 shows the network before synthesizing the top MUX.

Synthesis of Derived BS and FS Functions

At this point, both the set of BS functions $\vec{g}$ and the set of FS functions $\vec{h}$ have been derived, and the next task is to implement them as XAGs. For this purpose, we devised a near-MC-

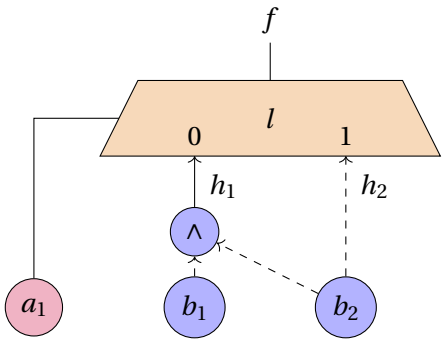


Figure 7.4: Partial circuit after deriving and synthesizing $\vec{h}$ (local) and $\vec{g}$ (joint). The top MUX l remains abstract at this stage; see Section 7.4.2.

minimum XAG synthesizer based on the SAT formulation introduced in [163]. The original formulation is limited to the synthesis of singular-output Boolean functions with up to five inputs. However, the cardinality of $\vec{g}$ and $\vec{h}$ is typically larger than one, indicating that the functionality of our BS/FS blocks is essentially *multi-output* functions defined respectively over the BS variables and FS variables. To accommodate this case, we extended the SAT-based approach to facilitate the synthesis of near-MC-minimal XAGs for multi-output functions by means of *incremental synthesis*: the solver targets one output function at a time, synthesizing an MC-minimal implementation in the sense that only the minimum number of additional AND nodes are introduced into the XAG synthesized for the outputs already considered. This approach achieves near-optimal quality while exhibiting far superior scalability compared to enforcing exact MC-optimality for multi-output functions, which scales poorly in practice.

One may question why the implementation quality of the BS and FS functions is measured in terms of MC. The rationale lies in the ownership of the underlying variables. As discussed in Section 7.3, BS variables are often *jointly derived*, which implies that most BS functions must be evaluated jointly; hence, their MC coincides with their JMC, making MC minimization directly relevant in this context. In contrast, FS variables are guaranteed by construction to be exclusively owned by a single party, ensuring that FS functions represent purely local computation. Nevertheless, for convenience and consistency, we also synthesize FS functions using the same incremental near-MC-minimum XAG synthesis engine.

With the BS and FS functions synthesized, the only remaining task to fully reconstruct the original function f is the synthesis of the top multiplexer l, which uses the outputs of the $|\vec{g}|$ BS functions to select among the outputs of the $|\vec{h}|$ FS functions. This final step is detailed in the next subsection.

7.4.2 Low-MC MUX Construction

Why target MC-optimality. As established in the previous subsection, the data signals (outputs of FS functions) belong to a single party, whereas the selecting signals (outputs of BS

functions) are typically *jointly derived*. In standard MUX realizations (e.g., $\neg \mathsf{sel} \cdot \mathsf{in}_0 + \mathsf{sel} \cdot \mathsf{in}_1$), data and select entangle from the first logic level, which makes it reasonable to treat the entire MUX as joint computation. Consequently, for MUX synthesis we have JMC = MC, and pursuing MC-optimal (or provably low-MC) constructions directly reduces the GC cost.

Base case: 2-to-1 MUX with one AND (MC/JMC = 1) Using the SAT-based MC-minimal XAG synthesis for single-output functions [163], a 2-to-1 MUX requires exactly one AND:

$$\begin{aligned} \mathrm{MUX}(\mathsf{sel}, \mathsf{in}_0, \mathsf{in}_1) &= \mathsf{sel}\ ?\ \mathsf{in}_1 : \mathsf{in}_0 \\ &= \big[(\mathsf{in}_0 \oplus \mathsf{in}_1) \wedge \mathsf{sel}\big] \oplus \mathsf{in}_0. \end{aligned}$$

Intuitively, if the data inputs are equal, the output ignores sel; if they differ, the output toggles by sel relative to in_0.

General case: n-to-1 MUX via binary composition A well-known decomposition builds an n-to-1 MUX as a full binary tree of 2-to-1 MUXes, requiring $(n-1)$ such nodes [1]. Substituting the one-AND realization for each 2-to-1 MUX yields an XAG with

$$\mathrm{MC} = \mathrm{JMC} = n - 1$$

for the top MUX. This construction is deterministic, avoids SAT solving, and therefore adds negligible synthesis time.

Implication for ACD-based flow. In our setting, l selects among $|\vec{h}|$ FS outputs using $|\vec{g}| = \lceil \log_2 |\vec{h}| \rceil$ control bits, so

$$\mathrm{JMC}(l) \ = \ |\vec{h}| - 1.$$

In the running example (Figure 7.4, $|\vec{h}| = 2$), $\mathrm{JMC}(l) = 1$, reproducing Figure 7.1b and halving JMC relative to Figure 7.1a.

Implication for our ACD-based flow Since the top MUX l selects among $|\vec{h}|$ FS outputs using $|\vec{g}| = \lceil \log_2 |\vec{h}| \rceil$ control signals, the above construction gives

$$\mathrm{JMC}(l) = |\vec{h}| - 1.$$

Applying this to the running example reproduces Figure 7.1 b, achieving a 50% reduction in JMC compared to the original implementation in Figure 7.1 a.

[1] The binary-tree composition of multiplexers is standard in digital design; any fan-in-n MUX can be realized by $(n-1)$ cascaded 2-to-1 MUXes organized as a balanced tree.

In summary, the techniques presented in this section — joint compute-oriented decomposition, near-MC-minimal synthesis of the derived BS and FS functions, and low-MC MUX construction — jointly constitute the sub-network resynthesis engine invoked in line 7 of Algorithm 7.1. This engine forms the computational core of our framework, enabling the systematic replacement of sub-networks with optimized XAG implementations that minimize jointly derived computation while preserving functional correctness.

7.5 Experimental Evaluation

To assess the effectiveness of our proposed optimization framework, we conducted a series of experiments on representative arithmetic benchmarks. The goal of this evaluation is to assess how the proposed ownership-aware analysis and its supporting logic optimization framework uncover circuit-level optimization opportunities that are complementary to, and extend beyond, conventional MC reduction.

7.5.1 Experimental Setup

All experiments were conducted on a Qualcomm Snapdragon X Plus machine with 16 GB RAM. All benchmarks were preprocessed using high-effort MC reduction techniques [171, 118]. For each benchmark, the optimization iteratively applies the framework (Algorithm 7.1) until no further improvement is found. The cut size bound κ is empirically set to 8, which offers a favorable trade-off between optimization granularity and runtime efficiency.

We report the following measures:

- **#ANDs**: number of AND gates in the baseline XAG after MC reduction;
- **#Joint ANDs (w/o opt.)**: number of AND gates identified as jointly derived under the ownership analysis introduced in Section 7.3.1;
- **#Joint ANDs (w. opt.)**: number of jointly derived AND gates after applying our optimization framework.
- **Red.**: relative reduction in garbled AND gates, i.e.,

$$\text{Red.} = \frac{\#\text{ANDs} - \#\text{Joint ANDs (w/ opt.)}}{\#\text{ANDs}};$$

- **Time**: end-to-end optimization time.

7.5.2 Benchmarks

We test our method on sixteen circuits drawn from three standard MPC-oriented benchmark suites — slightly deviating from those introduced in Section 2.3 for a more targeted evaluation — and organize them into two categories for analysis.

Benchmark	#ANDs	#Joint ANDs (w/o opt.)	#Joint ANDs (w. opt.)	Red. [%]	Time [s]
INT-add	128	128	128	0.00	1.44
INT-div	5 291	5 205	5 121	3.21	123.17
INT-mul	7 653	7 410	7 377	3.64	17.98
INT-eq	92	92	92	0.00	1.00
FP-add	5 141	4 938	4 824	6.17	15.20
FP-div	48 408	44 834	44 564	7.94	77.42
FP-mul	16 684	13 113	12 859	22.93	18.25
FP-eq	315	191	97	69.21	7.29

Table 7.1: Results on arithmetic and comparative benchmarks.

Arithmetic and comparative functions The first category includes seven benchmarks performing numerical or relational operations: 128-bit integer addition (`INT-add`), 64-bit integer division (`INT-div`), 64-bit integer multiplication (`INT-mul`), 32-bit integer equality (`INT-eq`), 64-bit floating-point addition (`FP-add`), 64-bit floating-point multiplication (`FP-mul`), and 64-bit floating-point equality (`FP-eq`). The integer circuits are taken from the EPFL combinational benchmark suite [5], while the floating-point circuits are taken from Nigel Smart's MPC Circuits collection [2]. All these benchmarks compute a relation over two numeric operands; accordingly, we assign one operand to each party, yielding a natural two-party ownership model.

Cryptographic and general-purpose kernels The second category comprises five circuits from the `TinyGarble` suite [166]: `AES-128`, `SHA3`, 32-bit `Hamming` distance, 32-bit `Encoder`, and the `LiteMIPS` processor core. For these benchmarks, we adopt the original input-ownership assignments provided with the suite.

While all experiments thus instantiate two-party computations, we emphasize that the proposed notion of JMC and the accompanying resynthesis and optimization framework are not limited to two-party scenarios, just as demonstrated throughout the chapter.

7.5.3 Results

Analysis of results on arithmetic and comparative benchmarks The first group of results (Table 7.1) exhibits two clear patterns: (1) Floating-point operators provide substantially more opportunities for reducing joint computation than their integer counterparts; and (2) Floating-point comparison benefits most from our optimization framework.

While integer operators are structurally dominated by joint dependencies, floating-point operators differ fundamentally. Despite the large circuit size, only a core portion of the computation, such as mantissa addition for `FP-add` or partial-product accumulation for

[2] Available at: https://nigelsmart.github.io/MPC-Circuits/

Benchmark	#ANDs	#Joint ANDs (w/o opt.)	#Joint ANDs (w. opt.)	Red. [%]	Time [s]
AES-128	6 446	5 153	5 153	20.06	1.61
SHA3	1 600	1 600	1 600	0.00	1.18
Hamming	48	48	48	0.00	0.92
Encoder	53	10	6	88.68	2.25
LiteMIPS	11 465	9 380	6 965	39.25	32.06

Table 7.2: Results on cryptographic and processor kernels.

FP-mul, genuinely mixes the two operands; much of the surrounding logic (e.g., extracting fields, preparing operands, post-processing results) acts on each operand independently. Our framework leverages this structure to isolate local regions and shrink the jointly derived portion. This yields meaningful reductions for FP-add, FP-div, and FP-mul, while integer operators remain comparatively rigid.

FP-eq exhibits the largest reduction in this group, because its behavior is driven primarily by *case analysis* rather than heavy two-operand arithmetic. Many outcomes, such as handling zeros, infinities, or NaNs, depend on only one operand, and even the remaining comparison logic is implemented with lightweight predicates. As a result, most of the computation is local, and the small joint core is aggressively compressed by our optimizer.

Analysis of results on cryptographic and general-purpose kernels The cryptographic benchmarks exhibit very limited potential for joint computation minimization. Designs such as AES-128 and SHA3 intentionally isolate their nonlinear operations: AES concentrates all nonlinearity in the S-boxes, and SHA3 derives it exclusively from the χ step. These components leave little scope for restructuring the circuit to further reduce joint computation.

The general-purpose kernels (Hamming, Encoder, LiteMIPS) contain richer mixtures of datapath and control logic, and their locality properties vary significantly. Hamming distance entangles the two inputs immediately through bitwise XOR, eliminating any chance for local computation. In contrast, the LiteMIPS processor exposes substantial regions of operand-local computation, such as instruction decoding and control steering, which translate into a 39.25% reduction in garbling cost: 18.19% comes from incorporating ownership awareness, and an additional 25.75% is achieved through the optimization framework.

Overall, these experiments demonstrate that our framework unlocks a new dimension of garbling cost reduction that is orthogonal to classical MC minimization. Because all input circuits have already undergone high-effort MC reduction, the total number of AND gates is hardly further irreducible; yet, by distinguishing local from joint computation, our method systematically reduces the number of jointly derived ANDs and thereby lowers the true cost of garbled evaluation. At the same time, the results confirm that the extent of achievable savings is inherently application-dependent.

7.6 Discussion

7.6.1 Application-Dependent Benefits

The experimental results highlight both the promise and the boundaries of our framework. As suggested by the analysis of the results reported in Tables 7.1 and 7.2, the practicality of our approach is inherently application-dependent. This aligns with the intuition that functions with richer structural diversity and more opportunities for variable partitioning expose greater potential for maximizing local computation, whereas structurally uniform functions such as integer addition remain bottlenecked by unavoidable joint operations.

7.6.2 Role of Ownership-Aware Analysis

Second, the correlation between the existence of local ANDs and successful JMC reduction underscores the importance of ownership-aware analysis. Just as *information-flow tracking* in hardware security propagates taint labels through gates to identify potential leakage points, our framework leverages ownership propagation to identify precisely where local processing can replace costly joint computation. This analogy highlights that JMC reduction is not merely a gate-count minimization exercise, but rather a finer-grained restructuring of the computational dependency graph.

7.6.3 A New Dimension of Secure Computation Optimization

Finally, the results reaffirm that our proposal opens a fundamentally new dimension of optimization in secure computation. While classical MC reduction, including Chapter 6, exhaustively removes non-linear gates, our perspective in this chapter explicitly distinguishes between locally derivable and jointly derived computation, which is complementary to prior works. This distinction enables systematic confinement of the MPC protocol to its essential core, reducing the garbling overhead without compromising correctness. In practice, this translates to more efficient deployments of secure computation frameworks in scenarios dominated by arithmetic kernels, where even modest reductions in jointly derived gates yield substantial performance gains.

7.6.4 Parallels with Information Flow Tracking

Beyond garbled circuits, the conceptual parallels with *gate-level information flow tracking* (GLIFT) [100] are particularly noteworthy. GLIFT was introduced as a methodology for tracking information flows through hardware circuits in order to detect malicious behaviors and ensure provable security guarantees [8, 99]. In practice, GLIFT operates by associating a *taint* label with each signal and propagating it through logic gates according to well-defined rules, such that if one input is tainted, the output is typically also tainted. This mechanism is strongly reminiscent of our ownership-aware propagation, where a node is deemed jointly

derived if any of its fanins are jointly derived. Here, jointly derived nodes in our setting correspond to tainted signals in GLIFT, while locally derivable nodes correspond to clean ones. This parallel suggests potential extensions of our methodology to GLIFT optimization, where systematic reduction of tainted gates and signals remains a largely unexplored problem [98]. Exploring such cross-fertilization offers a promising avenue for future research.

8 Conclusions and Outlook

This chapter reviews the core ideas and contributions developed across the thesis and reflects on their implications for the broader secure computation ecosystem. We begin with a compact summary of the five technical chapters — what problem each addresses, the key technical approach, and the measured impact. We then discuss overarching lessons that cut across secure-computation schemes, and close with higher-level remarks on standardization and cross-layer co-design that, in our view, are essential to make secure computation truly practical at scale.

8.1 Summary of Technical Contributions

Chapter 3(Beyond Depth: Joint MD/MC Synthesis for Leveled FHE). For leveled schemes (e.g., BFV/BGV), prior logic flows largely minimized *multiplicative depth* (MD) and treated the resulting increase in *multiplicative complexity* (MC) as a tolerable side effect. This chapter shows that this extensively-adopted MD-only optimization strategy typically leads to suboptimal leveled-FHE circuits. In contrast, we

(i) formalize an MD–MC composite objective guided by an empirical Pareto frontier,

(ii) develop an exact synthesis engine for small/medium cuts that yields provably optimum implementations under this objective, and

(iii) design a scalable, Boolean-correct rewriting framework that reduces peak MD *without* gratuitously increasing MC.

On standard benchmarks, this translates into consistent reductions in the cost model, more modest parameter sets, and tangible improvements in end-to-end evaluation latency.

Chapter 4 (Technology Mapping for TFHE: Gate-Set Design and Multi-Value PBS). This chapter targets fast-bootstrapping FHE for Boolean computation in a *single plaintext space*.

We revisit the common "two-input LUT mapping" formulation. Building on results that larger plaintext spaces permit higher-arity LUTs per *programmable bootstrap* (PBS), we provide

1. a constructive method to pack larger-input functions into a single PBS using Boolean properties, specifically, *symmetry* and *negacyclicity*, and
2. an MV-PBS–aware LUT mapper that increases input sharing to amortize PBS cost.

Compared to prior mappers, our designs invoke fewer PBS operations and deliver lower end-to-end latency across Boolean benchmarks.

Chapter 5 (Encoding Strategy Management for TFHE: An ESOP-Guided Approach). We enable *cross-space* TFHE-based Boolean function evaluation: exploit the binary space for cheap linear work (XOR) and jump via PBS to a larger plaintext space for compact non-linear LUTs, then return to binary as needed. The chapter introduces

(i) formal semantics for safe inter-space conversion,

(ii) a planner that places switching points and selects LUT granularities to minimize the total PBS (invoked for either computation or encoding-space switch) count,

(iii) an implementation that shows end-to-end speedups over single-space baselines on composite workloads.

Conceptually, Chapter 4 optimizes *within* a plaintext space; Chapter 5 orchestrates *across* plaintext spaces.

Chapter 6 (Ciphertext-Efficient Garbled Circuits via XOR–OneHot Graphs). In the *garbled circuit* (GC) protocol, XOR is free (via free-XOR), and the dominant cost is non-linearity. We show that AND is not necessarily the *ciphertext-optimal* carrier of non-linearity once translated to a garbled circuit. An analysis of primitive expressiveness and garbling cost motivates the *XOR–OneHot–inverter graph* (X1G) as a more cost-efficient representation. We provide

1. an optimal algebraic mapping from XAGs to X1Gs (preserving prior MC reductions while exploiting OneHot efficiency), and
2. X1G-specific optimizations that harvest savings unavailable to XAG-only flows.

The result is a consistent reduction in communicated ciphertexts for public benchmarks at negligible mapping overhead.

Chapter 7 (Redefining Cost in GC: Joint Multiplicative Complexity). Not every gate in a synthesized Boolean network needs to be garbled: nodes depending on a single party's inputs can be computed locally. We formalize *node ownership* and define *joint multiplicative complexity* (JMC) — the count of non-linear gates that *must* be garbled because their values depend jointly on multiple parties. We then introduce the first automated, general-purpose, gate-level framework to reduce JMC via ownership-aware subcircuit resynthesis. A decomposition-based engine provides low-JMC replacements; a global accept/reject strategy commits only rewrites that strictly decrease JMC in the whole circuit. This dimension is orthogonal to Chapter 6: together, they lower the cost per jointly evaluated non-linear gate and reduce the number of such gates, yielding end-to-end communication cost savings for the GC protocol.

8.2 Lessons Learned Across Projects

Two overarching lessons emerged repeatedly across the five projects:

(i) representation matters, and

(ii) cost models must match cryptographic reality.

To avoid repetition, chapter-specific limitations and prospects for extension are discussed in the closing section of each technical chapter; here, we focus only on cross-cutting insights that span projects.

8.2.1 Representation Matters

The right representation can cut costs immediately and unlock new optimization opportunities. Switching the logic representation can lower cost even before sophisticated optimization and can reveal transformations that were previously invisible. In GC, replacing the conventional XAG with the X1G (namely, pointing OneHot gates, rather than AND gates, as the nonlinearity provider in logic networks) shifts the basic nonlinearity unit: a three-input ONEHOT carries twice the nonlinearity of AND2 but garbles for the *same* ciphertext cost. As shown in Chapter 6, this yields

(i) *direct* garbling-cost reductions even under isomorphic mappings from XAG to X1G, and

(ii) *new* optimization passes (algebraic and don't-care rewrites, etc.) that are *specific* to ONEHOT semantics.

The representation must match the execution regime of the target scheme. A single cryptosystem can warrant different representations under different evaluation settings. In TFHE with a *single plaintext space* (every gate via PBS), a LUT network is the natural IR, and the

objective is to minimize LUTs/PBS calls (Chapter 4). In *multi-space* TFHE (cheap XOR in binary; expressive LUTs in larger spaces; PBS as a switch), an XAG is preferable: it cleanly exposes the linear/nonlinear boundary where switches occur and lets the optimizer place transitions to maximize linear work while minimizing PBS/conversion overhead (Chapter 5). In short, the chosen representation should mirror the scheme's *actual* execution model.

8.2.2 Cost Models Must Match Cryptographic Reality

Optimize against the true bottleneck, not a legacy proxy. As libraries and systems evolve, single-metric proxies can become misleading. In leveled FHE, minimizing *multiplicative depth* (MD) alone often inflates *multiplicative complexity* (MC) and slows execution. Chapter 3 shows that a composite MD-MC objective — guided by an empirical Pareto frontier — and algorithms that reduce MD without exploding MC deliver faster end-to-end evaluation, even when MD is not absolutely minimal.

Cost should further encode application semantics. The application context can change, which gates actually incur cryptographic cost. In GC, only non-linear gates whose values depend on multiple parties need garbling. By formalizing node ownership and minimizing *joint multiplicative complexity* (JMC), Chapter 7 reduces the number of gates that truly require GC, complementing representation-level gains (e.g., X1G). To conclude, effective objective functions must encode both protocol costs and application semantics, reflecting what must be computed securely.

8.3 Cross-Layer Reflections and a Call for Standardization

This thesis operates at one layer of a much broader stack (cf. Figures 1.2 and 1.3). Stepping back, we close with several high-level observations on how the community might accelerate progress by coordinating across layers — *applications, cryptographic constructions and libraries, compilers*, and *hardware* — so that work at any one layer composes smoothly with advances at the others.

A fragmented but fast-moving ecosystem. Unlike modern AI, where a set of dominant model families have catalyzed convergent compiler and hardware targets, secure computation (and FHE in particular) remains deliberately pluralistic: different scheme families (leveled vs. fast-bootstrapping; exact vs. approximate) offer complementary trade-offs and continue to evolve rapidly. This diversity is healthy, but it impedes large, long-horizon investments in downstream tooling and accelerators when interfaces and performance envelopes shift quickly. The risk is that downstream work becomes obsolete or even unreusable because assumptions were implicit and interfaces were ad hoc.

A pragmatic stance. We do *not* advocate waiting for a mythical "final" scheme before building the rest of the stack. Instead, we argue for stronger *standardization* and *modularity* so that improvements at one layer are quickly consumable at the others, and so that downstream artifacts (optimizers, kernels, accelerators) survive scheme evolution with minimal rework. Concretely, we propose the following community actions.

8.3.1 Recommendations for a Composable and Future-Proof Stack

Versioned cost model interfaces. Standardize an API whereby a crypto library exposes a *capability* and *cost profile* to compilers:

- supported operators;
- parameter ranges and security defaults;
- calibrated latency/energy models for primitive kernels.

Treat these as *versioned* contracts so that libraries can evolve while compilers remain compatible. Compilers then optimize against the declared costs rather than stale proxies. A promising community effort along these lines is `HEIR`, an open-sourced compiler for FHE that is being developed with participation from both academia and industry [3].

Reproducible benchmarks and reference workloads. Publish suites that exercise diverse patterns (dense linear algebra, control logic, etc.) with:

- *functional specifications* in the form of plaintext reference,
- *IR configurations*, and
- *ground-truth traces*, including operator counts, parameter sets, runtime on reference backends, etc.

A public "leaderboard" for end-to-end metrics, such as latency, throughput, energy, and memory footprint, would make improvements comparable across layers and prevent metric cherry-picking.

Hardware abstraction for FHE/MPC kernels. Create a minimal *cryptography-oriented kernel API* with clear data-layout contracts. This gives CPU/GPU/FPGA/ASIC implementers a stable target while allowing aggressive microarchitectural specialization underneath. In this way, compilers focus on mapping IR primitives to this API.

8.3.2 What This Means for Representations and Cost Models

The technical message of this thesis — *choose the representation that matches the execution regime; optimize against the cost that matches today's cryptographic reality* — extends naturally to standardization:

- IRs must carry the *semantics* that drive cost (e.g., switch sites (Chapter 5), ownership (Chapter 7)), not only structure.
- Cost models must be *adjustable while empirically calibrated* (Chapter 3), so that compilers can re-target as libraries and hardware improve.

With these enhancements, advances in cryptography (or more generally, mathematics), compilers, and hardware can be composed rather than re-implemented. This is how we collectively move secure computation from promising prototypes to dependable, everyday infrastructure.

Bibliography

[1] Martin Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. HomomorphicEncryption.org, 2018.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the Concrete Hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9 (3 2015), pp. 169–203. ISSN: 1862-2984. DOI: 10.1515/jmc-2015-0016.

[3] Asra Ali et al. "HEIR: A Universal Compiler for Homomorphic Encryption". In: *arXiv preprint 2508.11095* (2025). URL: https://arxiv.org/abs/2508.11095.

[4] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "Majority-Inverter Graph: A New Paradigm for Logic Optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.5 (2016), pp. 806–819. DOI: 10.1109/TCAD.2015.2488484.

[5] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "The EPFL Combinational Benchmark Suite". In: *Proceedings of International Workshop on Logic & Synthesis*. 2015.

[6] Luca Amarú et al. "Enabling Exact Delay Synthesis". In: *Proceedings of the International Conference on Computer-Aided Design*. 2017, pp. 352–359.

[7] David W. Archer et al. "RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications". In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2019, pp. 57–68. ISBN: 9781450368292. DOI: 10.1145/3338469.3358945.

[8] Armaiti Ardeshiricham et al. "Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design". In: *Design, Automation & Test in Europe Conference & Exhibition*. 2017, pp. 1691–1696. DOI: 10.23919/DATE.2017.7927266.

[9] Gilad Asharov et al. "Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE". In: *EUROCRYPT 2012*. Springer, 2012, pp. 483–501. ISBN: 978-3-642-29011-4.

[10] Robert L. Ashenhurst. "The Decomposition of Switch Functions". In: *Proceedings of the International Symposium on Theory of Switching Functions*. 1959, pp. 74–116.

[11] Louis J. M. Aslett, Pedro M. Esperança, and Chris C. Holmes. "A Review of Homomorphic Encryption and Software Tools for Encrypted Statistical Machine Learning". In: *arXiv preprint 1508.06574* (2015). URL: https://arxiv.org/abs/1508.06574.

[12] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. "Faster Homomorphic Encryption is not Enough: Improved Heuristic for Multiplicative Depth Minimization of Boolean Circuits". In: *Proceedings of the Cryptographers' Track at the RSA Conference*. Vol. 12006. Springer, 2020, pp. 345–363. DOI: 10.1007/978-3-030-40186-3_15.

[13] Gilles Audemard and Laurent Simon. *Glucose SAT solver 4.1*. 2017. URL: https://www.labri.fr/perso/lsimon/research/glucose/.

[14] Ahmad Al Badawi et al. "OpenFHE: Open-Source Fully Homomorphic Encryption Library". In: *IACR Cryptology ePrint Archive* (2022), p. 915. URL: http://eprint.iacr.org/2022/915.

[15] Rassul Bairamkulov and Giovanni De Micheli. "Superconductive Electronics: A 25-Year Review". In: *IEEE Circuits and Systems Magazine* 24.2 (2024), pp. 16–33. DOI: 10.1109/MCAS.2024.3376492.

[16] Marshall Ball, Tal Malkin, and Mike Rosulek. "Garbling Gadgets for Boolean and Arithmetic Circuits". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2016, pp. 565–577. ISBN: 9781450341394. DOI: 10.1145/2976749.2978410.

[17] Marshall Ball et al. "Garbled Neural Networks are Practical". In: *IACR Cryptology ePrint Archive* (2019), p. 338. URL: https://eprint.iacr.org/2019/338.

[18] Donald Beaver, Silvio Micali, and Phillip Rogaway. "The Round Complexity of Secure Protocols". In: *Proceedings of the Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 1990, pp. 503–513. ISBN: 0897913612. DOI: 10.1145/100216.100287.

[19] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. "Foundations of Garbled Circuits". In: *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2012, pp. 784–796. ISBN: 9781450316514. DOI: 10.1145/2382196.2382279.

[20] Mihir Bellare et al. "Efficient Garbling from a Fixed-Key Blockcipher". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2013, pp. 478–492. ISBN: 9780769549774. DOI: 10.1109/SP.2013.39.

[21] Loris Bergerat et al. "Parameter Optimization and Larger Precision for (T)FHE". In: *Journal of Cryptology* 36.3 (2023). ISSN: 0933-2790. DOI: 10.1007/s00145-023-09463-5.

[22] Loris Bergerat et al. "TFHE Gets Real: an Efficient and Flexible Homomorphic Floating-Point Arithmetic". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.2 (2025), pp. 126–162.

[23] Jonas Bertels et al. "Hardware Acceleration of FHEW". In: *IACR Cryptology ePrint Archive* (2023), p. 618. URL: http://eprint.iacr.org/2023/618.

[24] Fabian Boemer et al. “Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52”. In: *IACR Cryptology ePrint Archive* (2021), p. 420. URL: https://eprint.iacr.org/2021/420.

[25] Fabian Boemer et al. “nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data”. In: *Proceedings of the ACM International Conference on Computing Frontiers*. 2019, pp. 3–13. ISBN: 9781450366854. DOI: 10.1145/3310273.3323047.

[26] Fabian Boemer et al. “nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data”. In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2019, pp. 45–56. ISBN: 9781450368292. DOI: 10.1145/3338469.3358944.

[27] Peter Bogetoft et al. “A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation”. In: *Financial Cryptography and Data Security*. Springer, 2006, pp. 142–147. ISBN: 978-3-540-46256-9.

[28] Nicolas Bon, David Pointcheval, and Matthieu Rivain. “Optimized Homomorphic Evaluation of Boolean Functions”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.3 (2024), pp. 302–341.

[29] Keith Bonawitz et al. “Practical Secure Aggregation for Privacy-Preserving Machine Learning”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2017, pp. 1175–1191. DOI: 10.1145/3133956.3133982.

[30] Keith Bonawitz et al. “Towards Federated Learning at Scale: System Design”. In: *Proceedings of the SysML Conference*. 2019.

[31] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. “Large FHE Gates from Tensored Homomorphic Accumulator”. In: *AFRICACRYPT*. Springer, 2018, pp. 217–251. ISBN: 978-3-319-89339-6.

[32] Jean-Philippe Bossuat et al. “Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys”. In: *EUROCRYPT*. Springer, 2021, pp. 587–617. ISBN: 978-3-030-77870-5.

[33] Joan Boyar and René Peralta. “Tight Bounds for the Multiplicative Complexity of Symmetric Functions”. In: *Theoretical Computer Science* 396.1 (2008), pp. 223–246. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2008.01.030.

[34] Joan Boyar, René Peralta, and Denis Pochuev. “On the Multiplicative Complexity of Boolean Functions over the Basis $(\wedge, \oplus, 1)$”. In: *Theoretical Computer Science* 235.1 (2000), pp. 43–57. DOI: 10.1016/S0304-3975(99)00182-6.

[35] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *Innovations in Theoretical Computer Science*. ACM, 2012, pp. 309–325. DOI: 10.1145/2090236.2090262.

[36] Daniel Brand. “Redundancy and Don’t Cares in Logic Synthesis”. In: *IEEE Transactions on Computers* C-32.10 (1983), pp. 947–952. DOI: 10.1109/TC.1983.1676139.

[37] Robert K. Brayton and Alan Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool”. In: *Proceedings of the International Conference on Computer Aided Verification*. Vol. 6174. Springer, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[38] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* C-35 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[39] Çagdas Çalik, Meltem Sönmez Turan, and René Peralta. “The Multiplicative Complexity of 6-variable Boolean Functions”. In: *Cryptography and Communications* 11.1 (2019), pp. 93–107. DOI: 10.1007/S12095-018-0297-2.

[40] Sergiu Carpov. “A Fast Heuristic for Mapping Boolean Circuits to Functional Bootstrapping”. In: *IACR Cryptology ePrint Archive* (2024), p. 1204. URL: https://eprint.iacr.org/2024/1204.

[41] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. “A Multi-start Heuristic for Multiplicative Depth Minimization of Boolean Circuits”. In: *Proceedings of the International Workshop on Combinatorial Algorithms*. Vol. 10765. Springer, 2017, pp. 275–286. DOI: 10.1007/978-3-319-78825-8_23.

[42] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. “Armadillo: A Compilation Chain for Privacy Preserving Applications”. In: *Proceedings of the International Workshop on Security in Cloud Computing*. ACM, 2015, pp. 13–19. DOI: 10.1145/2732516.2732520.

[43] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. “New Techniques for Multi-value Input Homomorphic Evaluation and Applications”. In: *The Cryptographers’ Track at the RSA Conference*. Springer, 2019, pp. 106–126. ISBN: 978-3-030-12611-7. DOI: 10.1007/978-3-030-12612-4_6.

[44] Sergiu Carpov et al. “Practical Privacy-Preserving Medical Diagnosis Using Homomorphic Encryption”. In: *Proceedings of the IEEE International Conference on Cloud Computing*. IEEE Computer Society, 2016, pp. 593–599. DOI: 10.1109/CLOUD.2016.0084.

[45] Henry Carter et al. “Secure Outsourced Garbled Circuit Evaluation for Mobile Devices”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 289–304. ISBN: 978-1-931971-03-4.

[46] Julien de Castelnau, Mingfei Yu, and Giovanni De Micheli. “Cut Tracing with E-Graphs for Boolean FHE Circuit Synthesis”. In: *arXiv preprint 2506.12883* (2025). URL: https://arxiv.org/abs/2506.12883.

[47] Francesco Castro, Donato Impedovo, and Giuseppe Pirlo. “An Efficient and Privacy-Preserving Federated Learning Approach Based on Homomorphic Encryption”. In: *IEEE Open Journal of the Computer Society* 6 (2025), pp. 336–347. DOI: 10.1109/OJCS.2025.3536562.

[48] Gizem S. Çetin et al. “Depth Optimized Efficient Homomorphic Sorting”. In: *Proceedings of the International Conference on Cryptology and Information Security in Latin America*. Vol. 9230. Springer, 2015, pp. 61–80. DOI: 10.1007/978-3-319-22174-8_4.

[49] Dake Chen et al. “RNA-ViT: Reduced-Dimension Approximate Normalized Attention Vision Transformers for Latency Efficient Private Inference”. In: *IEEE/ACM International Conference on Computer Aided Design*. 2023, pp. 1–9. DOI: 10.1109/ICCAD57390.2023.10323702.

[50] Feng Chen et al. “Perfectly Secure and Efficient Two-Party Electronic-Health-Record Linkage”. In: *IEEE Internet Computing* 22.2 (2018), pp. 32–41. DOI: 10.1109/MIC.2018.112102542.

[51] Hao Chen, Kim Laine, and Rachel Player. “Simple Encrypted Arithmetic Library - SEAL v2.1”. In: *IACR Cryptology ePrint Archive* (2017), p. 224. URL: https://eprint.iacr.org/2017/224.

[52] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *ASIACRYPT*. 2017, pp. 409–437.

[53] Seonyoung Cheon et al. “DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024. ISBN: 978-1-939133-44-1.

[54] Ilaria Chillotti, Marc Joye, and Pascal Paillier. “Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks”. In: *Cyber Security Cryptography and Machine Learning*. Springer, 2021, pp. 1–19. ISBN: 978-3-030-78086-9.

[55] Ilaria Chillotti et al. “Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE”. In: *ASIACRYPT*. Springer, 2021, pp. 670–699. ISBN: 978-3-030-92078-4.

[56] Ilaria Chillotti et al. “Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE”. In: *ASIACRYPT*. Springer, 2021, pp. 670–699. ISBN: 978-3-030-92077-7. DOI: 10.1007/978-3-030-92078-4_23.

[57] Ilaria Chillotti et al. “TFHE: Fast Fully Homomorphic Encryption Over the Torus”. In: *Journal of Cryptology* 33 (2020), pp. 34–91.

[58] Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. “Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures”. In: *arXiv preprint 2407.13055* (2025). URL: https://arxiv.org/abs/2407.13055.

[59] Wutichai Chongchitmate and Rafail Ostrovsky. “Circuit-Private Multi-key FHE”. In: *Public-Key Cryptography*. Springer, 2017, pp. 241–270. ISBN: 978-3-662-54388-7.

[60] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. “Civitas: Toward a Secure Voting System”. In: *IEEE Symposium on Security and Privacy*. 2008, pp. 354–368. DOI: 10.1109/SP.2008.32.

[61] Luca Colombo, Alessandro Falcetta, and Manuel Roveri. “Training Encrypted Neural Networks on Encrypted Data with Fully Homomorphic Encryption”. In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2024, pp. 64–75. ISBN: 9798400712418. DOI: 10.1145/3689945.3694802.

[62] Jason Cong and Yuzheng Ding. "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Sesigns". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.1 (1994), pp. 1–12. DOI: 10.1109/43.273754.

[63] Jason Cong, Chang Wu, and Yuzheng Ding. "Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution". In: *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 1999, pp. 29–35. DOI: 10.1145/296399.296425.

[64] Ana Costache and Nigel P. Smart. "Which Ring Based Somewhat Homomorphic Encryption Scheme is Best?" In: *Proceedings of the Cryptographers' Track at the RSA Conference*. Vol. 9610. Springer, 2016, pp. 325–340. DOI: 10.1007/978-3-319-29485-8_19.

[65] Andrea Costamagna and Giovanni De Micheli. "Accuracy Recovery: A Decomposition Procedure for the Synthesis of Partially-Specified Boolean Functions". In: *Integration, the VLSI Journal* C (2023). ISSN: 0167-9260. DOI: 10.1016/j.vlsi.2022.12.008.

[66] Eric Crockett, Chris Peikert, and Chad Sharp. "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1020–1037. ISBN: 9781450356930. DOI: 10.1145/3243734.3243828.

[67] Herbert Allen Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand Reinhold Inc., 1962. ISBN: 978-0442017941.

[68] Wei Dai and Berk Sunar. "cuHE: A Homomorphic Encryption Accelerator Library". In: *Cryptography and Information Security in the Balkans*. Springer, 2016, pp. 169–186. ISBN: 978-3-319-29172-7.

[69] Roshan Dathathri et al. "CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019, pp. 142–156. DOI: 10.1145/3314221.3314628.

[70] Roshan Dathathri et al. "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 546–561. ISBN: 9781450376136. DOI: 10.1145/3385412.3386023.

[71] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[72] Deniel Demmler, Thomas Schneider, and Michael Zohner. "ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *Network and Distributed System Security Symposium*. 2015. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/08_2_1.pdf.

[73] Marten van Dijk et al. "Fully Homomorphic Encryption over the Integers". In: *EUROCRYPT*. Springer, 2010, pp. 24–43. ISBN: 978-3-642-13190-5.

[74] Josep Domingo-Ferrer et al. “Privacy-Preserving Cloud Computing on Sensitive Data: A Survey of Methods, Products and Challenges”. In: *Computer Communications* 140.C (2019), pp. 38–60. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2019.04.011.

[75] Léo Ducas and Daniele Micciancio. “FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second”. In: *EUROCRYPT*. 2015, pp. 617–640.

[76] Austin Ebel, Karthik Garimella, and Brandon Reagen. “Orion: A Fully Homomorphic Encryption Framework for Deep Learning”. In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2025, pp. 734–749. ISBN: 9798400710797. DOI: 10.1145/3676641.3716008.

[77] Colin R. Edwards. “The Application of the Rademacher-Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis”. In: *IEEE Transactions on Computers* 24.1 (1975), pp. 48–62. DOI: 10.1109/T-C.1975.224082.

[78] Niklas Eén and Niklas Sörensson. “MiniSat: An Extensible SAT-solver”. In: *Proceedings of the Theory and Applications of Satisfiability Testing*. 2004, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[79] Hassan Takabi Ehsan Hesamifard and Mehdi Ghasemi. “CryptoDL: Deep Neural Networks over Encrypted Data”. In: *arXiv preprint 1711.05189* (2017). URL: https://arxiv.org/abs/1711.05189.

[80] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. “SEALion: a Framework for Neural Network Inference on Encrypted Data”. In: *arXiv preprint 1904.12840* (2019). URL: https://arxiv.org/abs/1904.12840.

[81] Junfeng Fan and Frederik Vercauteren. “Somewhat Practical Fully Homomorphic Encryption”. In: *IACR Cryptology ePrint Archive* (2012), p. 144. URL: http://eprint.iacr.org/2012/144.

[82] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 2009, pp. 169–178. ISBN: 9781605585062. DOI: 10.1145/1536414.1536440.

[83] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *CRYPTO*. Springer, 2013, pp. 75–92. ISBN: 978-3-642-40041-4.

[84] Craig Gentry et al. “Outsourcing Private RAM Computation”. In: *IEEE Annual Symposium on Foundations of Computer Science*. 2014, pp. 404–413. DOI: 10.1109/FOCS.2014.50.

[85] Ran Gilad-Bachrach et al. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: *Proceedings of the International Conference on Machine Learning*. 2016, pp. 201–210.

[86] Shruthi Gorantala et al. *A General Purpose Transpiler for Fully Homomorphic Encryption*. Tech. rep. Google LLC, 2021.

[87] Eiichi Goto and Hidetoshi Takahashi. “Some Theorems Useful in Threshold Logic for Enumerating Boolean Functions”. In: *Proceedings of the International Federation for Information Processing Congress*. 1962, pp. 747–752.

[88] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. “SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks”. In: *Proceedings on Privacy Enhancing Technologies* 3 (2023), pp. 154–172.

[89] Charles Gouert and Nektarios Georgios Tsoutsos. “Romeo: Conversion and Evaluation of HDL Designs in the Encrypted Domain”. In: *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*. IEEE Press, 2020. ISBN: 9781450367257.

[90] Zhenyu Guan et al. “AutoHoG: Automating Homomorphic Gate Design for Large-Scale Logic Circuit Evaluation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43.7 (2024), pp. 1971–1983. ISSN: 0278-0070. DOI: 10.1109/TCAD.2024.3357598.

[91] Winston Haaswijk et al. “SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.4 (2020), pp. 871–884. DOI: 10.1109/TCAD.2019.2897703.

[92] Shai Halevi and Victor Shoup. “Algorithms in HElib”. In: *CRYPTO*. Springer, 2014, pp. 554–571. ISBN: 978-3-662-44371-2.

[93] Shai Halevi and Victor Shoup. “Design and implementation of HElib: a homomorphic encryption library”. In: *IACR Cryptology ePrint Archive* (2020), p. 1481. URL: https://eprint.iacr.org/2020/1481.

[94] Kyoohyung Han and Dohyeong Ki. “Better Bootstrapping for Approximate Homomorphic Encryption”. In: *Proceedings of the Cryptographers’ Track at the RSA Conference*. Springer, 2020, pp. 364–390. ISBN: 978-3-030-40186-3.

[95] Thomas Häner and Mathias Soeken. “Lowering the T-depth of Quantum Circuits via Logic Network Optimization”. In: *ACM Transactions on Quantum Computing* 3.2 (2022). DOI: 10.1145/3501334.

[96] Todd Haselton. *Credit Reporting Firm Equifax Says Cybersecurity Incident Could Potentially Affect 143 Million US Consumers*. CNBC. Sept. 7, 2017. URL: https://www.cnbc.com/2017/09/07/credit-reporting-firm-equifax-says-cybersecurity-incident-could-potentially-affect-143-million-us-consumers.html.

[97] Edward Helmore. *Genetic Testing Firm 23andMe Admits Hackers Accessed DNA Data of 7M Users*. The Gaurdian. Dec. 5, 2023. URL: https://www.theguardian.com/technology/2023/dec/05/23andme-hack-data-breach.

[98] Wei Hu et al. “An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2021), pp. 1010–1038. DOI: 10.1109/TCAD.2020.3047976.

[99] Wei Hu et al. “Detecting Hardware Trojans with Gate-Level Information-Flow Tracking”. In: *Computer* 49.8 (2016), pp. 44–52. DOI: 10.1109/MC.2016.225.

[100] Wei Hu et al. “Theoretical Fundamentals of Gate Level Information Flow Tracking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (2011), pp. 1128–1140. DOI: 10.1109/TCAD.2011.2120970.

[101] Zhicong Huang et al. “Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference”. In: *USENIX Security Symposium*. 2022, pp. 809–826. ISBN: 978-1-939133-31-1.

[102] Lei Jiang, Qian Lou, and Nrushad Joshi. “MATCHA: a fast and energy-efficient accelerator for fully homomorphic encryption over the torus”. In: *Proceedings of the ACM/IEEE Design Automation Conference*. 2022, pp. 235–240. ISBN: 9781450391429. DOI: 10.1145/3489517.3530435.

[103] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. “GAZELLE: A Low Latency Framework for Secure Neural Network Inference”. In: *USENIX Security Symposium*. 2018, pp. 1651–1669. ISBN: 978-1-939133-04-5.

[104] Sangpyo Kim et al. “BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption”. In: *Proceedings of the Annual International Symposium on Computer Architecture*. 2022, pp. 711–725. ISBN: 9781450386104. DOI: 10.1145/3470496.3527415.

[105] Christian Knabenhans et al. “vFHE: Verifiable Fully Homomorphic Encryption”. In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2024, pp. 11–22. ISBN: 9798400712418. DOI: 10.1145/3689945.3694806.

[106] Donald E. Knuth. “The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation”. In: Addison-Wesley Professional, 2013.

[107] Vladimir Kolesnikov and Ranjit Kumaresan. “Improved OT Extension for Transferring Short Secrets”. In: *CRYPTO*. Springer, 2013, pp. 54–70. ISBN: 978-3-642-40084-1.

[108] Vladimir Kolesnikov and Thomas Schneider. “Improved Garbled Circuit: Free XOR Gates and Applications”. In: *Proceedings of the International Colloquium on Automata, Languages and Programming, Part II*. Springer, 2008, pp. 486–498. ISBN: 9783540705826. DOI: 10.1007/978-3-540-70583-3_40.

[109] Alexander I. Kornilov and Tatiana Yu Isaeva. “Circuit Depth Optimization by BDD Based Function Decomposition”. In: *Logic and Architecture Synthesis: State-of-the-art and novel approaches*. Springer, 1995, pp. 64–69. ISBN: 978-0-387-34920-6. DOI: 10.1007/978-0-387-34920-6_6.

[110] Aleksandar Krastev et al. “A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption”. In: *Proceedings of the ACM Programming Languages* 8 (2024). DOI: 10.1145/3656382.

[111] *Lattigo v6*. Online:https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA. Aug. 2024.

[112] Dongkwon Lee et al. “Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting”. In: *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 2020, pp. 503–518. DOI: 10.1145/3385412.3385996.

[113] Wouter Legiest et al. “Neural Network Quantisation for Faster Homomorphic Encryption”. In: *IEEE International Symposium on On-Line Testing and Robust System Design*. 2023, pp. 1–3. DOI: 10.1109/IOLTS59296.2023.10224890.

[114] Tancrède Lepoint and Pascal Paillier. “On the Minimal Number of Bootstrappings in Homomorphic Circuits”. In: *Financial Cryptography and Data Security*. Vol. 7862. Springer, 2013, pp. 189–200. DOI: 10.1007/978-3-642-41320-9_13.

[115] Long Li et al. “ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference”. In: *Proceedings of the ACM/IEEE International Symposium on Code Generation and Optimization*. 2025, pp. 193–208. ISBN: 9798400712753. DOI: 10.1145/3696443.3708924.

[116] Yehuda Lindell. “Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries”. In: *CRYPTO*. Springer, 2013, pp. 1–17. ISBN: 978-3-642-40084-1.

[117] Yehuda Lindell and Benny Pinkas. “Secure two-party computation via cut-and-choose oblivious transfer”. In: *Proceedings of the Conference on Theory of Cryptography*. Springer, 2011, pp. 329–346. ISBN: 9783642195709.

[118] Hsiao-Lun Liu et al. “A Don’t-Care-Based Approach to Reducing the Multiplicative Complexity in Logic Networks”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.11 (2022), pp. 4821–4825. ISSN: 0278-0070. DOI: 10.1109/TCAD.2022.3147444.

[119] Yan Liu et al. “ReSBM: Region-based Scale and Minimal-Level Bootstrapping Management for FHE via Min-Cut”. In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2025, pp. 924–939. ISBN: 9798400706981. DOI: 10.1145/3669940.3707276.

[120] Qian Lou et al. “vFHE: Verifiable Fully Homomorphic Encryption with Blind Hash”. In: *arXiv preprint 2303.08886* (2023). URL: https://arxiv.org/abs/2303.08886.

[121] Valavan Manohararajah, Stephen D. Brown, and Zvonko G. Vranesic. “Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.11 (2006), pp. 2331–2340. DOI: 10.1109/TCAD.2006.882119.

[122] Dewmini Sudara Marakkalage et al. “Three-Input Gates for Logic Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.10 (2021), pp. 2184–2188. DOI: 10.1109/TCAD.2020.3032625.

[123] Kotaro Matsuoka et al. “Towards Better Standard Cell Library: Optimizing Compound Logic Gates for TFHE”. In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. Association for Computing Machinery, 2021, pp. 63–68. ISBN: 9781450386562. DOI: 10.1145/3474366.3486927.

[124] D. Michael Miller and Mathias Soeken. “An Algorithm for Linear, Affine and Spectral Classification of Boolean Functions”. In: *Advanced Boolean Techniques*. Springer, 2020, pp. 195–215. ISBN: 978-3-030-20323-8. DOI: 10.1007/978-3-030-20323-8_9.

[125] Alan Mishchenko, Robert Brayton, and Satrajit Chatterjee. “Boolean Factoring and Decomposition of Logic Networks”. In: *IEEE/ACM International Conference on Computer-Aided Design*. 2008, pp. 38–44. DOI: 10.1109/ICCAD.2008.4681549.

[126] Alan Mishchenko and Robert K. Brayton. “SAT-Based Complete Don’t-Care Computation for Network Optimization”. In: *Design, Automation & Test in Europe Conference & Exhibition*. 2005, pp. 412–417. DOI: 10.1109/DATE.2005.264.

[127] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. “DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis”. In: *Proceedings of the 43rd Annual Design Automation Conference*. 2006, pp. 532–535. ISBN: 1595933816. DOI: 10.1145/1146909.1147048.

[128] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. “Improvements to Technology Mapping for LUT-Based FPGAs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 240–253. DOI: 10.1109/TCAD.2006.887925.

[129] Alan Mishchenko and Marek Perkowski. “Fast Heuristic Minimization of Exclusive-Sums-of-Products”. In: *International Reed-Muller Workshop*. 2001.

[130] Alan Mishchenko et al. “Combinational and Sequential Mapping with Priority Cuts”. In: *IEEE/ACM International Conference on Computer-Aided Design*. 2007, pp. 354–361. DOI: 10.1109/ICCAD.2007.4397290.

[131] Alan Mishchenko et al. “Delay Optimization Using SOP Balancing”. In: *IEEE/ACM International Conference on Computer-Aided Design*. 2011, pp. 375–382. DOI: 10.1109/ICCAD.2011.6105357.

[132] Pratyush Mishra et al. “Delphi: A Cryptographic Inference Service for Neural Networks”. In: *USENIX Security Symposium*. 2020, pp. 2505–2522. ISBN: 978-1-939133-17-5.

[133] Johannes Mono, Kamil Kluczniak, and Tim Güneysu. “Improved Circuit Synthesis with Amortized Bootstrapping for FHEW-like Schemes”. In: *IACR Cryptology ePrint Archive* (2023), p. 1223. URL: http://eprint.iacr.org/2023/1223.

[134] Johannes Mono et al. “Finding and Evaluating Parameters for BGV”. In: *AFRICACRYPT*. 2023, pp. 370–394.

[135] Christian Mouchet et al. “Helium: Scalable MPC among Lightweight Participants and under Churn”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2024, pp. 3038–3052. ISBN: 9798400706363. DOI: 10.1145/3658644.3670346.

[136] Christian Mouchet et al. “Multiparty Homomorphic Encryption from Ring-Learning-with-Errors”. In: *Proceedings on Privacy Enhancing Technologies* (4 2021), pp. 291–311.

[137] Saburo Muroga. “Threshold Logic and Its Applications”. In: John Wiley & Sons, 1971. ISBN: 0471625302.

[138] Cody D. Murray and R. Ryan Williams. “On the (Non) NP-Hardness of Computing Circuit Complexity”. In: *Proceedings of the Conference on Computational Complexity*. 2015, pp. 365–380. ISBN: 9783939897811.

[139] Moni Naor, Benny Pinkas, and Reuban Sumner. “Privacy preserving auctions and mechanism design”. In: *Proceedings of the ACM Conference on Electronic Commerce*. Association for Computing Machinery, 1999, pp. 129–139. ISBN: 1581131763. DOI: 10.1145/336992.337028.

[140] Shintaro Narisada et al. “Time-Memory Trade-off Algorithms for Homomorphically Evaluating Look-up Table in TFHE”. In: *Proceedings of the Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2024, pp. 1–10. ISBN: 9798400712418. DOI: 10.1145/3689945.3694801.

[141] National Institute of Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*. Tech. rep. FIPS PUB 197. U.S. Department of Commerce, 2001. URL: https://doi.org/10.6028/NIST.FIPS.197.

[142] Marie Paindavoine and Bastien Vialla. “Minimizing the Number of Bootstrappings in Fully Homomorphic Encryption”. In: *Proceedings of the International Conference on Selected Areas in Cryptography*. Vol. 9566. Springer, 2015, pp. 25–43. DOI: 10.1007/978-3-319-31301-6_2.

[143] Arpita Patra et al. “ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation”. In: *USENIX Security Symposium*. 2021, pp. 2165–2182. ISBN: 978-1-939133-24-3.

[144] Marek Perkowski and Malgorzata Chrzanowska-Jeske. “An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions”. In: *IEEE International Symposium on Circuits and Systems*. 1990, pp. 1652–1655. DOI: 10.1109/ISCAS.1990.112455.

[145] Rishabh Poddar et al. “Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics”. In: *USENIX Security*. 2021, pp. 2129–2146. ISBN: 978-1-939133-24-3.

[146] David Du Pont et al. “Hardware Acceleration of the Prime-Factor and Rader NTT for BGV Fully Homomorphic Encryption”. In: *IEEE Symposium on Computer Arithmetic*. 2024, pp. 1–8. DOI: 10.1109/ARITH61463.2024.00011.

[147] Deevashwer Rathee et al. “CrypTFlow2: Practical 2-Party Secure Inference”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 325–342. ISBN: 9781450370899. DOI: 10.1145/3372297.3417274.

[148] Deevashwer Rathee et al. “MPC-Minimized Secure LLM Inference”. In: *arXiv preprint 2408.03561* (2024). URL: https://arxiv.org/abs/2408.03561.

[149] Oded Regev. “On Lattices, Learning with Errors, Random Linear Codes, and Cryptography”. In: *Jounal of the ACM* 56.6 (2009). ISSN: 0004-5411. DOI: 10.1145/1568318.1568324.

[150] M. Sadegh Riazi et al. “Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications”. In: *Proceedings of the 2018 Asia Conference on Computer and Communications Security*. 2018, pp. 707–721. ISBN: 9781450355766. DOI: 10.1145/3196494.3196522.

[151] M. Sadegh Riazi et al. “HEAX: An Architecture for Computing on Encrypted Data”. In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1295–1309. ISBN: 9781450371025. DOI: 10.1145/3373376.3378523.

[152] M. Sadegh Riazi et al. “MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation”. In: *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2019, pp. 198–207. DOI: 10.1109/HST.2019.8740831.

[153] Heinz Riener et al. “Exact Synthesis of ESOP Forms”. In: *Advanced Boolean Techniques*. Springer, 2020, pp. 177–194. ISBN: 978-3-030-20322-1.

[154] Heinz Riener et al. “On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis”. In: *Design, Automation & Test in Europe Conference & Exhibition*. 2019, pp. 1649–1654. DOI: 10.23919/DATE.2019.8715185.

[155] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. “On Data Banks and Privacy Homomorphisms”. In: *Foundations of Secure Computation* 4.11 (1978), pp. 169–180.

[156] Nikola Samardzic et al. “CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data”. In: *Proceedings of the Annual International Symposium on Computer Architecture*. 2022, pp. 173–187. ISBN: 9781450386104. DOI: 10.1145/3470496.3527393.

[157] Nikola Samardzic et al. “F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption”. In: *Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 238–252. ISBN: 9781450385572. DOI: 10.1145/3466752.3480070.

[158] Tsutomu Sasao. “And-Exor Expressions and their Optimization”. In: *Logic Synthesis and Optimization*. Springer, 1993, pp. 287–312. ISBN: 978-1-4615-3154-8. DOI: 10.1007/978-1-4615-3154-8_13.

[159] Adi Shamir. “How to Share a Secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176.

[160] Claude E. Shannon. “A Symbolic Analysis of Relay and Switching Circuits”. In: *Transactions of the American Institute of Electrical Engineers* 57.12 (1938), pp. 713–723. DOI: 10.1109/T-AIEE.1938.5057767.

[161] Zach Simas. *Unpacking the MOVEit Breach: Statistics and Analysis*. Emsisoft | Cybersecurity Blog. July 18, 2023. URL: https://www.emsisoft.com/en/blog/44123/unpacking-the-moveit-breach-statistics-and-analysis/.

[162] Sujoy Sinha Roy et al. “FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data”. In: *IEEE International Symposium on High Performance Computer Architecture*. 2019, pp. 387–398. DOI: 10.1109/HPCA.2019.00052.

[163] Mathias Soeken. “Determining the Multiplicative Complexity of Boolean Functions using SAT”. In: *IACR Cryptology ePrint Archive* (2020), p. 530. URL: https://eprint.iacr.org/2020/530.

[164] Mathias Soeken et al. “Practical Exact Synthesis”. In: *Design, Automation & Test in Europe Conference & Exhibition*. 2018, pp. 309–314. DOI: 10.23919/DATE.2018.8342027.

[165] Mathias Soeken et al. “The EPFL Logic Synthesis Libraries”. In: *arXiv preprint 1805.05121* (2022). URL: https://arxiv.org/abs/1805.05121.

[166] Ebrahim M. Songhori et al. “TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 411–428. ISBN: 9781467369497. DOI: 10.1109/SP.2015.32.

[167] Ross Tate et al. “Equality Saturation: A New Approach to Optimization”. In: *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2009, pp. 264–276. DOI: 10.1145/1480881.1480915.

[168] Alessandro Tempia Calvino and Giovanni De Micheli. “Technology Mapping Using Multi-Output Library Cells”. In: *IEEE/ACM International Conference on Computer Aided Design*. 2023, pp. 1–9. DOI: 10.1109/ICCAD57390.2023.10323999.

[169] Alessandro Tempia Calvino et al. “A Versatile Mapping Approach for Technology Mapping and Graph Optimization”. In: *Asia and South Pacific Design Automation Conference*. 2022, pp. 410–416. DOI: 10.1109/ASP-DAC52403.2022.9712552.

[170] Alessandro Tempia Calvino et al. “Enhancing Delay-Driven LUT Mapping With Boolean Decomposition”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44.3 (2025), pp. 1017–1030. DOI: 10.1109/TCAD.2024.3457378.

[171] Eleonora Testa et al. “A Logic Synthesis Toolbox for Reducing the Multiplicative Complexity in Logic Networks”. In: *Design, Automation & Test in Europe Conference & Exhibition*. 2020, pp. 568–573. DOI: 10.23919/DATE48585.2020.9116467.

[172] Eleonora Testa et al. “Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications”. In: *Proceedings of the Annual Design Automation Conference*. ACM, 2019, pp. 1–6. DOI: 10.1145/3316781.3317893.

[173] Grigori S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: Classical Papers on Computational Logic*. Springer, 1983, pp. 466–483. DOI: 10.1007/978-3-642-81955-1_28.

[174] Meltem Sönmez Turan and René Peralta. “The Multiplicative Complexity of Boolean Functions on Four and Five Variables”. In: *IACR Cryptology ePrint Archive* (2015), p. 848. URL: http://eprint.iacr.org/2015/848.

[175] Michiel Van Beirendonck et al. “FPT: A Fixed-Point Accelerator for Torus Fully Homomorphic Encryption”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 741–755. ISBN: 9798400700507. DOI: 10.1145/3576915.3623159. URL: https://doi.org/10.1145/3576915.3623159.

[176] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. “SoK: Fully Homomorphic Encryption Compilers”. In: *IEEE Symposium on Security and Privacy*. 2021, pp. 1092–1108. DOI: 10.1109/SP40001.2021.00068.

[177] Alexander Viand et al. “HECO: Fully Homomorphic Encryption Compiler”. In: *Proceedings of the USENIX Conference on Security Symposium*. 2023, pp. 4715–4732. ISBN: 978-1-939133-37-3.

[178] Nikolaj Volgushev et al. “Conclave: Secure Multi-Party Computation on Big Data”. In: *Proceedings of the EuroSys Conference*. 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303982.

[179] Jelle Vos, Mauro Conti, and Zekeriya Erkin. “Oraqle: A Depth-Aware Secure Computation Compiler”. In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2024, pp. 43–50.

[180] Feng Wang et al. “Dual-Output LUT Merging during FPGA Technology Mapping”. In: *Proceedings of the International Conference on Computer-Aided Design*. Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415617.

[181] Zeyu Wang and Makoto Ikeda. “High-Throughput Key Switching Accelerator for Homomorphic Encryption”. In: *International Conference on IC Design and Technology*. 2023, pp. 100–103. DOI: 10.1109/ICICDT59917.2023.10332291.

[182] Zeyu Wang and Makoto Ikeda. “High-Throughput Privacy-Preserving GRU Network with Homomorphic Encryption”. In: *International Joint Conference on Neural Networks*. 2023, pp. 1–9. DOI: 10.1109/IJCNN54540.2023.10191194.

[183] Zeyu Wang and Makoto Ikeda. “Toward Bootstrapping-Free Homomorphic Encryption-Based GRU Network for Text Classification”. In: *IEEE Access* 12 (2024), pp. 94008–94017. DOI: 10.1109/ACCESS.2024.3422455.

[184] Henry S. Warren. “Hacker’s Delight, Second Edition”. In: Pearson Education, 2013. ISBN: 0-321-84268-5.

[185] Tommy White et al. “FHE-Booster: Accelerating Fully Homomorphic Execution with Fine-tuned Bootstrapping Scheduling”. In: *IEEE International Symposium on Hardware Oriented Security and Trust*. 2023, pp. 293–303. DOI: 10.1109/HOST55118.2023.10132930.

[186] Claire Wolf. *Yosys Open SYnthesis Suite*. https://yosyshq.net/yosys/.

[187] Tianshi Xu et al. “Breaking the Layer Barrier: Remodeling Private Transformer Inference with Hybrid CKKS and MPC”. In: *Proceedings of the USENIX Conference on Security Symposium*. 2025. ISBN: 978-1-939133-52-6.

[188] Congguang Yang, Maciej Ciesielski, and Vigyan Singhal. “BDS: A BDD-Based Logic Optimization System”. In: *Proceedings of the Annual Design Automation Conference*. Association for Computing Machinery, 2000, pp. 92–97. ISBN: 1581131879. DOI: 10.1145/337292.337323.

[189] Wenlong Yang, Lingli Wang, and Alan Mishchenko. “Lazy man’s logic synthesis”. In: *Proceedings of the International Conference on Computer-Aided Design*. Association for Computing Machinery, 2012, pp. 597–604. ISBN: 9781450315739.

[190] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets”. In: *Annual Symposium on Foundations of Computer Science*. 1986. DOI: 10.1109/SFCS.1986.25.

[191] Mingfei Yu, Gabrielle De Micheli, and Giovanni De Micheli. “Making the Best Switch: Encoding Strategy Management for Efficient TFHE Circuit Evaluation”. In: *IEEE/ACM International Conference on Computer Aided Design*. 2025.

[192] Mingfei Yu and Giovanni De Micheli. “Faster Homomorphic Operations and Beyond: Expediting Homomorphic Computation via Boolean Circuit Optimization”. In: *Journal of Cryptology* (2025). accepted.

[193] Mingfei Yu and Giovanni De Micheli. “Generating Lower-Cost Garbled Circuits: Logic Synthesis Can Help”. In: *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust*. 2023, pp. 304–314. DOI: 10.1109/HOST55118.2023.10133215.

[194] Mingfei Yu, Dewmini Sudara Marakkalage, and Giovanni De Micheli. “Garbled Circuits Reimagined: Logic Synthesis Unleashes Efficient Secure Computation”. In: *Cryptography* 7.4 (2023). ISSN: 2410-387X. DOI: 10.3390/cryptography7040061.

[195] Mingfei Yu and Giovanni De Micheli. “Striving for Both Quality and Speed: Logic Synthesis for Practical Garbled Circuits”. In: *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. IEEE, 2023, pp. 1–9. DOI: 10.1109/ICCAD57390.2023.10323660.

[196] Mingfei Yu et al. “On the Synthesis of High-performance Homomorphic Boolean Circuits”. In: *Proceedings of the ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2024, pp. 51–63.

[197] Samee Zahur, Mike Rosulek, and David Evans. “Two Halves Make a Whole”. In: *EUROCRYPT*. Springer, 2015, pp. 220–250. ISBN: 978-3-662-46803-6.

[198] Zama. *Concrete: TFHE Compiler that Converts Python Programs into FHE Equivalent*. https://github.com/zama-ai/concrete. 2022.

[199] Zama. *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*. 2022. URL: https://github.com/zama-ai/tfhe-rs.

Mingfei Yu

Email : mingfeiyuee@gmail.com
Mobile : (+41)772673314

Education

EPFL — Lausanne, Switzerland
Ph.D. in Electrical Engineering — *Apr. 2022 - Dec. 2025*

University of Tokyo — Tokyo, Japan
M.Eng. in Electrical Engineering and Information Systems — *Sept. 2019 - Mar. 2022*

Zhejiang University — Hangzhou, China
B.Eng. in Electrical Engineering — *Sept. 2015 - June 2019*

Research Experience

Integrated System Laboratory, EPFL — Lausanne, Switzerland
Ph.D. Candidate · Advisor: Prof. Giovanni De Micheli — *Apr. 2022 – Dec. 2025*

- Efficient Synthesis of High-Performance TFHE Circuits.
- Accelerating Leveled Homomorphic Computation via Circuit Optimization.
- Practical Garbled Circuits Generation.

Computer Security Laboratory, CWI — Amsterdam, the Netherlands
Research Intern · Advisor: Dr. Chenglu Jin, Prof. Marten van Dijk — *May 2025 – Aug. 2025*

- Parallelism-Aware Secure Multi-Party Computation via Garbled Circuits.

University of Tokyo — Tokyo, Japan
Research Assistant · Advisor: Prof. Masahiro Fujita — *Feb. 2020 – Mar. 2022*

- Parallel Scheduling of Modern Deep Learning Implementations.
- Low-Precision Quantization for the BERT Model.

AIST-UTokyo AI chip Design Open Innovation Laboratory — Tokyo, Japan
Research Assistant · Advisor: Prof. Masahiro Fujita, Dr. Shinichi O'uchi — *Sept. 2020 – Mar. 2021*

- An FPGA-based, Flexibly-Pipelined CNN Accelerator Design.

Selected Publications

- **M. Yu** and G. De Micheli, "Faster Homomorphic Operations and Beyond: Expediting Homomorphic Computation via Boolean Circuit Optimization," *Journal of Cryptology*, 2025.
- **M. Yu**, G. De Micheli, and G. De Micheli, "Making the Best Switch: Encoding Strategy Management for Efficient TFHE Circuit Evaluation," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2025.
- **M. Yu**, M. Soeken, and G. De Micheli, "A New Perspective of Constructing Resource-Efficient Data-Lookup Quantum Oracles," *IEEE Quantum Computing and Engineering (QCE)*, 2025.
- **M. Yu**, A. Tempia Calvino, M. Soeken, and G. De Micheli, "Back-end-aware Fault-tolerant Quantum Oracle Synthesis," *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2025 (**Best Paper Nomination**).
- **M. Yu**, S. Carpov, A. Tempia Calvino, and G. De Micheli, "On the Synthesis of High-Performance Homomorphic Boolean Circuits," *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2024.
- **M. Yu**, D.S. Marakkalage, and G. De Micheli, "Garbled Circuits Reimagined: Logic Synthesis Unleashes Efficient Secure Computation," *Cryptography*, 2023.
- **M. Yu** and G. De Micheli, "Striving for Both Quality and Speed: Logic Synthesis for Practical Garbled Circuits," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023 (**Best Paper Award**).
- **M. Yu** and G. De Micheli, "Generating Low-Cost Garbled Circuits: Logic Synthesis Can Help," *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023.

- C. Meng, **M. Yu**, H. Wang, W. Burleson, and G. De Micheli, "RareLS: Rarity-Reducing Logic Synthesis for Mitigating Hardware Trojan Threats, " *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024.
- R. Bairamkulov, **M. Yu**, and G. De Micheli, "Unleashing the Power of T1-cells in SFQ Arithmetic Circuits," *IEEE/ACM Design Automation Conference (DAC)*, 2024.

Work Experience

EPFL — Lausanne, Switzerland
Teaching Assistant · CS-472: Design Technologies for Integrated Systems — *Fall 2024*

EPFL — Lausanne, Switzerland
Teaching Assistant · COM-112: Object-Oriented Programming — *Spring 2023, 2025*

EPFL — Lausanne, Switzerland
Teaching Assistant · CS-119: Information, Communication and Calculation — *Fall 2022, 2023*

NVIDIA — Shanghai, China
Research Intern · System-ASIC Group — *Mar. 2021 – June 2021*

Languages

Mandarin (native) , English (fluent) , Japanese (N1)

Technical Skills

Programming Languages: C/C++ , Unix Script , Rust
Hardware Description Languages: Verilog , Chisel , SpinalHDL
EDA Tools: Catapult (Mentor Graphics) , Design Compiler (Synopsys) , Vivado (AMD) , VCS (Synopsys)

Selected Awards

The Teaching Assistant Annual Award at EPFL	Dec. 2024
IEEE/ACM William J. McCalla ICCAD Best Paper Award (Front-end)	Nov. 2023
Top X in CAD Contest at ICCAD: Learning Arithmetic Operations from Gate-Level Circuit	Oct. 2022
IPSJ T-SLDM Best Paper Award	Sept. 2022
Excellent Graduation Thesis at the University of Tokyo	Mar. 2022
4th Place in CAD Contest at ICCAD: Functional ECO with Behavioral Change Guidance	Oct. 2021
Best Poster Award, School of Computing and Communication Systems at the University of Tokyo	July 2021
3rd Place in Programming Contest at IWLS: Circuit Learning	July 2021
IEICE VLD Excellent Student Author Award	Mar. 2021
4th Place in CAD Contest at ICCAD: Equivalence Checking for Logic Circuits with Don't Cares	Sept. 2020
Best Video Award at DAC Young Fellow Program	Aug. 2020
DAC Young Fellow Program	Aug. 2020
1st Place in Programming Contest at IWLS: Circuit Learning	July 2020