View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

User-Controlled Computations in Untrusted Computing Environments

by

Dhinakaran Vinayagamurthy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science (Quantum Information)

Waterloo, Ontario, Canada, 2019

© Dhinakaran Vinayagamurthy 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Prof. David Evans
Professor, Department of Computer Science, University of Virginia.

Supervisor(s): Prof. Sergey Gorbunov
Assistant Professor, David R. Cheriton School of Computer Science.
Prof. David Jao
Associate Professor, Department of Combinatorics and Optimization.

Internal Members: Prof. Florian Kerschbaum
Associate Professor, David R. Cheriton School of Computer Science.
Prof. Michele Mosca
Professor, Department of Combinatorics and Optimization.

Internal-External Member: Prof. Douglas Stebila
Associate Professor, Department of Combinatorics and Optimization.

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Computing infrastructures are challenging and expensive to maintain. This led to the growth of cloud computing with users renting computing resources from centralized cloud providers. There is also a recent promise in providing decentralized computing resources from many participating users across the world. The *compute on your own server* model hence is no longer prominent. But, traditional computer architectures, which were designed to give a complete power to the owner of the computing infrastructure, continue to be used in deploying these new paradigms. This forces users to completely trust the infrastructure provider on all their data. The cryptography and security community research two different ways to tackle this problem. The first line of research involves developing powerful cryptographic constructs with formal security guarantees. The primitive of functional encryption (FE) formalizes the solutions where the clients do not interact with the sever during the computation. FE enables a user to provide computation-specific secret keys which the server can use to perform the user specified computations (and only those) on her encrypted data. The second line of research involves designing new hardware architectures which remove the infrastructure owner from the trust base. The solutions here tend to have better performance but their security guarantees are not well understood. This thesis provides contributions along both lines of research. In particular,

- We develop a (single-key) functional encryption construction where the size of secret keys do not grow with the size of descriptions of the computations, while also providing a tighter security reduction to the underlying computational assumption. This construction supports the computation class of branching programs. Previous works for this computation class achieved either short keys or tighter security reductions but not both.

- We formally model the primitive of *trusted hardware* inspired by Intel's Software Guard eXtensions (SGX). We then construct an FE scheme in a strong security model using this trusted hardware primitive. We implement this construction in our system Iron and evaluate its performance. Previously, the constructions in this model relied on heavy cryptographic tools and were not practical.

- We design an encrypted database system StealthDB that provides complete SQL support. StealthDB is built on top of Intel SGX and designed with the usability and security limitations of SGX in mind. The StealthDB implementation on top of Postgres achieves practical performance (30% overhead over plaintext evaluation) with strong leakage profile against adversaries who get snapshot access to the memory

of the system. It achieves a more gradual degradation in security against persistent adversaries than the prior designs that aimed at practical performance and complete SQL support.

We finally survey the research on providing security against quantum adversaries to the building blocks of SGX.

Acknowledgements

The years of grad school has provided me a great learning experience with some highs and lows. A lot of people have helped me during this journey and this is an opportunity for me to consolidate my gratitude.

The one with the greatest impact is Sergey Gorbunov. From providing advice on having a healthy life as a graduate student during my masters years, to shaping my research direction in PhD based on my long-term career goals. An input from him that I am happy to have developed (I hope) is the importance of having the big picture of the research area in mind, and with this, trying to fit whatever I read into the big picture. I had always admired researchers with a good breadth in their expertise and hoped to be one myself. His contempt towards saying something is "outside one's expertise" and his insistence on "learning as you need" has helped me take a small step in obtaining the breadth. Thanks for everything, Sergey! I wish I were a faster learner during these years so that I could have obtained a better depth in these topics and made more out of the interactions with him.

The primary reason for my smooth PhD journey in Waterloo is David Jao. Despite his research interests being quite different from what I worked on, David had been gracious enough to completely fund my PhD and sometimes slightly *bend the rules* in funding to let me follow my research interests without any restrictions. He has always been available for any questions or concerns I had. I also thank David for all the help during my job search. Many thanks to Michele Mosca for providing a complete research freedom while financially supporting me, along with David, and for all the advice whenever I ask for one. I hope I have delivered enough for the high expectations David and Mike have on me.

It is always fun to work with Alexey Gribov. Thanks Alex for the collaboration during StealthDB and for leading the development part of it! Thanks to Ben Fisch and Dan Boneh for bringing in their version of the FE-HW model and results, and for the great discussions during the merge which created Iron.

Thanks to Florian Kerschbaum for his comments on StealthDB and the great advice regarding working in industry research. Thanks to Douglas Stebila for providing a detailed feedback on my thesis and to David Evans for reading my thesis and participating in my defense.

The CrySP group has a great culture and every CrySPer has the opportunity to grow towards being a complete crypto, security and privacy researcher. Thanks to Ian Goldberg for setting up and providing a great environment for CrySPers. Thanks guys for the reading groups and all the feedback on my research and talks. Many thanks

to my officemate Sainath for all the great conversations and discussions. His curiosity in science and technology meant that, providing a little background, I can have deep technical discussions on my research and sometimes also get suggestions from him, despite him being an experimental physicist. Thanks to Filip and Alex Norton for all the interesting crypto discussions. Thanks to Goutam for making StealthDB much better to work with.

During my gradschool, I had two great internships at Microsoft Research. Thanks to Nishanth Chandran for hosting me in Bangalore, and Kapil Vaswani in Cambridge. Thanks to Nishanth also for all the help during job search.

Thanks to the great Charlie Rackoff for advising me during my PhD time in Toronto, for asking me to question more before I agree on any statement, and to take more care on the precision of definitions. Long before that, thanks to C. Pandu Rangan, Sharmila Deva Selvi and Sree Vivek for introducing me to the field of cryptography and research, and to Vinod Vaikuntanathan for increasing my crypto knowledge manifold.

After recovering from some tough moments towards the end of my masters, the great group of friends at Waterloo and Toronto ensured a fun-filled PhD life. Thanks to Sharat, Cedric and recently Abhinav (at Waterloo) and Jai, Subbu and Bharathwaj (at Toronto) for being great housemates and for tolerating the high variance in the taste of the food that I cook. The long discussions with Sharat on any news or topic of interest/curiosity are always fun and informative. I owe a lot to Cedric for getting me into running and for converting me from someone who rarely runs to a decent paced long distance runner (in 5 months). Thanks also to Jonathan, Karthik and Jimmy for all the conversations during the long runs. Thanks to Cedric also for always being available and attentive for a practice talk at home before any talk of mine. The weekly Friday late evening gathering with my house mates and Hemant, Jimit, Priya, Anirudh, Retnika, Archana, Akshay, Varuna, Paolos and Mali (sorted by frequency of attendance!) is always full of fun conversations. Many times I felt that I had enough fun for the week during those hours that I don't need more of the weekend. Thanks guys! Squash is one of the things that helped me have fun even during tough moments. The weekly few hours of squash is something I always look forward to. Thanks Hemant, Abhinav, Jimit, Akshay and Cedric (in Waterloo), and David, Mika, Norman, Sergey and Venkatesh (in Toronto) for enabling this. Out of context, thanks to Hemant also for helping with some details of Postgres during StealthDB and to Anirudh for all the discussion and ongoing collaboration that is not part of this thesis. Thanks to Arumani for checking on me once in a while from India!

Above all, I cannot thank enough my parents Jothimani and Vinayagamurthy and my grandparents for their incomparable love and support. Their faith in my abilities and their endless encouragement are crucial over the years in whatever I have accomplished.

Dedication

To my parents, Jothimani and Vinayagamurthy.

Table of Contents

List of Tables xiv

List of Figures xv

1 Introduction 1

1.1 Cryptographic solutions . . . 2

1.2 Trusted hardware based solutions . . . 4

2 Controlled computations from Lattice-based Cryptography 7

2.1 Lattice-based cryptography . . . 7

2.2 Attribute-based encryption . . . 8

2.2.1 Our Results . . . 9

2.2.2 Applications . . . 12

2.2.3 Other Related Work . . . 12

2.2.4 Extensions . . . 13

2.2.5 Organization of this chapter . . . 13

2.3 Preliminaries . . . 13

2.3.1 Lattice Preliminaries . . . 14

2.3.2 Attribute-Based Encryption definition . . . 16

2.3.3 Branching Programs . . . 17

2.4 Our Evaluation Algorithms . . . 18

2.4.1 Basic Homomorphic Operations 18
2.4.2 Our Public Key Evaluation Algorithm 21
2.4.3 Our Encoding Evaluation Algorithm 22
2.4.4 Our Simulated Public Key Evaluation Algorithm 24
2.5 Our Attribute-Based Encryption 26
2.5.1 Correctness 29
2.5.2 Security Proof 29
2.6 Parameter Selection 34
2.7 Single-key Functional Encryption with short secret keys 36
2.7.1 Definitions 36
2.7.2 Construction 38
2.8 Conclusion and next steps 39

3 A Formal Introduction to SGX **41**

3.1 Intel SGX Background 41
3.1.1 Isolation. 42
3.1.2 Sealing. 42
3.1.3 SGX Attestation 42
3.1.4 SGX TCB. 43
3.1.5 SGX side-channel attacks and defenses 44
3.2 Formal Models and Definitions 45
3.2.1 Formal HW model 45
3.3 HW correctness and security definitions 48
3.3.1 Local attestation unforgeability 48
3.3.2 Remote attestation unforgeability 49
3.4 Differences between HW and Intel SGX 51

4 Functional Encryption from SGX **53**

4.1 The need for practical solutions for Functional Encryption 53

4.2 Our contributions 54

4.2.1 Construction overview 55

4.3 Related Work 56

4.4 System Design 57

4.4.1 Architecture overview 57

4.4.2 FE Protocols 60

4.5 Implementation and evaluation 62

4.5.1 Implemented ECALLS 63

4.5.2 Performance evaluation 65

4.6 Formalization of Iron 68

4.6.1 Formal definition of Functional Encryption 68

4.6.2 Crypto primitive definitions 71

4.6.3 FE Formal construction 74

4.7 Security 78

4.7.1 Security proof 79

4.8 FE construction in the stronger security model 87

4.8.1 Security overview 90

4.9 Extensions and Future Work 92

5 StealthDB: A Scalable Encrypted Database on SGX **94**

5.1 Introduction 94

5.1.1 Our contributions 96

5.2 Intel SGX 98

5.3 Platform Overview 98

5.3.1 Usage Model. 98

5.3.2 Threat Model 99

5.4 Designing an Encrypted DB 99
5.4.1 Design Goals 99
5.4.2 Designing an Encrypted DB from SGX 101
5.5 Architecture 104
5.5.1 Database creation 104
5.5.2 DBMS Initialization 105
5.5.3 Client authentication 107
5.5.4 Query execution 108
5.5.5 Encrypting indexes 110
5.5.6 Extensions 111
5.6 Security 111
5.6.1 Leakage profile 112
5.6.2 Security of K during StealthDB execution 116
5.7 Concrete leakage profiles 118
5.8 Implementation and Performance 119
5.8.1 Implementation details 119
5.8.2 Performance evaluation 120
5.9 Related Work 121
5.10 Conclusion 123

6 Quantum resistance of SGX **125**
6.1 Introduction 125
6.2 Cryptography used by SGX core 126
6.3 Enhanced Privacy ID (EPID) 126
6.3.1 Definition 127
6.3.2 Construction based on one-way functions 129
6.3.3 Construction based on lattices 132
6.3.4 Quantum security 133

6.4 AES and MAC . . . 134
6.5 Cryptography for applications in SGX . . . 134
6.6 Conclusion . . . 135

7 The path ahead **136**
7.1 Cryptographic constructions . . . 136
7.2 FE from trusted hardware . . . 137
7.3 Encrypted databases using trusted hardware supporting complete SQL . . . 138

References **140**

List of Tables

5.1 Latency statistics of TPC-C requests, ms 121

6.1 Post-quantum EPID and EPID-like schemes. 133

List of Figures

4.1 IRON Architecture and Protocol Flow 59
4.2 FE.Setup and FE.Keygen run time. 66
4.3 Breakdown of FE.Decrypt run times for each of our SGX-FE implementations of IBE, ORE, and 3DNF. 66
4.4 Comparison of decryption times and ciphertext sizes for the SGX-FE implementation of IBE, ORE, 3DNF to cryptographic implementations. . . 67
4.5 Comparison of time for decrypting 10^3 ciphertext tuples using the SGX-FE implementation of IBE, ORE, 3DNF vs cryptographic implementations from pairings and mmaps respectively. 68
4.6 Q_{KME} 75
4.7 Q_{DE} 77
4.8 $Q_{FE}(P)$ 78
4.9 Q_{KME} II 89
4.10 Q_{DE} II 90

5.1 High-level architecture overview of StealthDB 97
5.2 Three alternative design choices for an encrypted database with SGX. . . . 101
5.3 Initialization time comparing in memory and in enclave deserialization for different dataset sizes. 102
5.4 Latency to execute random binary tree searches comparing different approaches. 103
5.5 StealthDB architecture. 104

5.6 Definition of *enc_int4* 105
5.7 Create table 105
5.8 The authentication protocol of StealthDB 106
5.9 Operator = for *enc_int4*. Here, *enc_int4_eq* will call the Ops enclave to decrypt the input, check their equality and output the result. 109
5.10 Security definition for an encrypted database system using trusted hardware. 113
5.11 Example of a new function definition in stealthdb.sql 119
5.12 Example of new defined function implementation in stealthdb.c 119
5.13 TPC-C benchmarking throughput for running under Postgres and StealthDB with different scale factors 121
5.14 Average latency and standard deviation for TPC-C requests under Postgres and StealthDB. 122

7.1 Summary of designs in this thesis. 137

Chapter 1

Introduction

Traditionally, users stored and processed their data either in their personal computers or in the computers and servers owned by the organizations they hired to manage their data. Hence, the traditional architectures of computers were designed to give a complete authority to the *owners* of the system to manage *their* system. A system administrator has the full power to monitor and control the programs and the associated data with the help of the operating system and the hypervisor. Security tools were developed to make use of the monitoring to assist the admin in improving the security of the system.

Over the last decade, computing paradigms have shifted from owning and using personal computers or server machines to pay-per-use remote rented computing environments. This shift started with the success of cloud computing with the cloud providers renting out computing resources of varying capabilities, adaptively as required, with various software packages installed and ready-to-use. With the emergence of decentralized computing environments a.k.a. blockchains, user's computing can be done by any set of participating users around the world. A shift further away from using owned computing resources. The consequence of all these:

> User data no longer remains in its owner's computers.

Traditional computer architectures continue to be used though. Infrastructure providers like the cloud providers are completely trusted to not (mis)use user data. In addition to trusting providers to handle entrusted data in a secure way, there is a need for trust that the data is protected and isolated from other potentially malicious users who are sharing

the provided resources. Given this scenario, protecting users' data is an essential problem in these potentially untrusted computing environments.

Digital data exists in one of the following three states: at-rest, in-transit and in-use. There are different ways of defining what it means for data to be in a particular state, depending on how much we zoom in to the computing architecture. But at a high level, data-at-rest refers to the data that is in a storage location and not being processed, data-in-transit refers to the data that is being transferred between two entities, and data-in-use refers to the data that is being processed to get some useful information. The advancements in cryptography and security have led to good solutions for securing data-in-transit (e.g. TLS) and data-at-rest (e.g. AES). But the security of data-in-use is a more nuanced and challenging problem. A user should be able to provide *controlled access* of her data to the computing infrastructure such that even a malicious administrator of the infrastructure cannot learn any information about her data when processing her request. And the user can be guaranteed that the obtained result corresponds to the requested computation. What is an ideal situation that we could strive to achieve here?

> An infrastructure provider should act just as a *dumb* infrastructure provider renting out resources like networking, storage, memory and computing power, and be *completely oblivious* to how the user uses these resources. And this is enforced.

There are other desirable properties for users. For example, if a user outsourced a computation, allowing the user to be offline while the computation is executed remotely is beneficial. There are two major lines of research that tackle this problem that are introduced in the following sections.

1.1 Cryptographic solutions

One line of work involves designing cryptographic techniques for controlled computations with strong formal security guarantees. The notion of functional encryption formally studies this problem. Cryptography for securing cloud computation has also been studied under notions like fully homomorphic encryption [95] and secure multi-party computation [210, 100], but they typically expect the client to be online to perform or complete the computation. A Functional Encryption (FE) scheme lets an user encrypt her data under a master key[1] and provide the encrypted data to the cloud provider. Later,

[1] A user can generate as many master keys as needed to encrypt different datasets.

whenever she wants the cloud provider to perform a computation on her data she can derive computation-specific secret keys which when provided enable the cloud provider to perform those computations. The computation secret keys are independent of the data and can be used to compute on any data encrypted with a specific master key using which the computation secret keys are derived. The security of functional encryption ensures that the cloud provider learns no information about the user's data other than the result of the computations requested by the user. There exist reasonably efficient solutions (upto an order of magnitude overhead over unencrypted computations) for very specific computations like inner-products [3, 8]. But extremely high performance overheads are incurred when the solution is general and not restricted to these specific class of computations [147]. Previous works also have large sizes for the computation-specific secret key. In particular, their size grew either polynomially with the size of the circuit [107] or rely on stronger security assumptions [42].

Single-key functional encryption for branching programs

The first result in this thesis works on reducing the key size while relying on weaker security assumptions for the computation class of polynomial length branching programs [30]. The construction is based on the Learning With Errors (LWE) assumption [95] whose average-case hardness is based on the worst-case hardness of well-studied computational problems in lattices. These problems are also believed to be hard for quantum computers to solve. The modulus of the group or the ring used in the construction determine the approximation factors for the underlying lattice problems during the security reduction of the LWE assumption to those lattice problems. Hence, the modulus influences the security of the construction. We obtain a (single-key) functional encryption scheme with short keys and small modulus (and hence polynomial approximation factors).

Theorem 1.1.1. *(informal) There exists a single-key secure functional encryption scheme for a family of length-L branching programs P with small secret keys based on the security of the Learning With Errors assumption with polynomial approximation factors for the lattice problems. We have $|\mathsf{sk}_P| = |P| + poly(\lambda, \log L)$ and the modulus $q = \mathrm{poly}(\lambda, L)$, where λ is the security parameter.*

The single-key condition means that secret key for only one program can be provided per colluding set of adversaries. But this secret key can be used to compute on any number of datasets encrypted with the same master key. The computation class of polynomial length branching programs is equivalent to the computation class NC^1 which includes the

computations that can represented as boolean circuits of depth logarithmic in the input size [30]. Prior work based on standard cryptographic assumptions either required keys proportional to the size of the program or have super-polynomial approximation factors.

The core of our result is an attribute-based encryption (ABE) scheme [189, 113] for the same family of length-L branching programs P with short secret keys and polynomial approximation factors. Previous works for this computation class either had $|\mathsf{sk}_P| = |P| \cdot \mathrm{poly}(\lambda)$ [106, 107] or $q = \mathrm{poly}(\lambda, L^{\log L})$ [42]. In addition to security, a smaller modulus also leads directly to efficiency improvements. ABE is interesting on its own with applications to fine-grained access control [13, 150, 176]. ABE is also used for constructing the powerful primitive of reusable garbled circuits [104]. This work is presented in Chapter 2.

1.2 Trusted hardware based solutions

The other line of research involves designing secure computing architectures that minimize the trust assumptions on the owner and the administrator of the computing infrastructure through enforced isolation along with (basic) encryption and integrity protection. This line of research over the last decade has resulted in solutions from both academia [152, 199, 200, 65, 88, 156, 76] and industry [15, 115, 128, 158] with varying trust assumptions and security guarantees. The recent academic proposals [76] achieve strong security guarantees. And there has been an ongoing effort on developing an open-source full stack implementation building on [76]. Currently, Intel's Software Guard eXtensions (SGX) [158] is the only industry solution that has been aimed at providing a strong isolation against the administrator of the computer, and that has been extensively studied. In SGX, the processor is trusted and the memory regions used by different programs are encrypted with different keys derived from a master key that resides in the processor embedded during the manufacturing phase. This way, the hypervisor and the operating systems in the cloud computer need not be trusted by a remote user using the computer when she trusts the hardware manufacturer and the correctness of the trusted hardware design.

Iron: Functional encryption from trusted hardware

The second contribution of this thesis is a formal modeling of the trusted hardware primitives like SGX and a provably secure construction of functional encryption from our primitive of trusted hardware. Our construction of FE satisfies a simulation-based security model which is a strong version of security for FE. Previous works on the construction of

FE utilizing trusted hardware used simulatable hardware "tokens" which do not model a practical trusted hardware architecture [72].[2]

Theorem 1.2.1. *(informal) There exists a simulation-secure functional encryption scheme assuming a trusted hardware scheme, secure signature scheme and a secure public key encryption scheme.*

We build our FE system IRON and evaluate its performance instantiating the trusted hardware primitive with SGX. This work is presented in Chapters 3 and 4 with Chapter 3 explaining the modeling of trusted hardware and Chapter 4 presenting our construction of FE.

Our construction in IRON though assumes an *ideal* behavior of the trusted hardware that a program running inside the trusted region does not leak any information. In practice, SGX has been found to incur some security and usability limitations. There are many side-channels [75, 208, 207, 56] found in the SGX implementation which reveal significant information about the data being processed when care is not taken. Also, there is a sharp degradation in performance [174] as the size of the trusted memory region used grows above 128 MB. A significant portion of this performance bottleneck is due to Merkle-tree management for providing integrity and this seems to be inherent in SGX-style trusted hardware architectures.

StealthDB: A scalable and complete encrypted database system

The next result in this thesis designs an encrypted transactional database system StealthDB built on top of SGX that scales to large datasets. The design of StealthDB is aimed at providing practical performance and minimal changes to the database system to which security properties are augmented. StealthDB provides a strong leakage profile against semi-honest attackers who get a snapshot access to memory and a graceful degradation of security against attackers with a persistent access. The evaluation of our design built on Postgres shows that:

> StealthDB can process transactional (OLTP) queries with a 30% reduction in throughput and ≈ 1 ms overhead in latency over an unencrypted DBMS with > 10M total rows (or 2 GB plaintext) of a standard benchmarking (TPC-C warehouse) database for scale factor $W = 16$.

StealthDB is presented in Chapter 5.

[2] In [72], hardware tokens need to be transmitted as the computation secret keys to the computing infrastructure. More details in Chapter 4.

Quantum resistance of SGX

Cryptographic primitives like symmetric-key encryption, hash functions and a variant of group signatures [54] assume key roles in the security of SGX. The advent of quantum computers requires strengthening of these cryptographic primitives. The group signature scheme [54] relies on Diffie-Hellman like assumptions which can be broken easily by quantum algorithms [196]. The current designs of block ciphers and hash functions admit no special structure to be heavily exploited by quantum computers better than the quadratic speedup of [116]. But the symmetric primitives built on top of them do need careful analysis for quantum security and sometimes require stronger assumptions and modifications for quantum security. Chapter 6 of this thesis surveys the work on the quantum resistance of the cryptography used in SGX and presents some open research questions to be solved in this space pertaining to the applications in trusted hardware.

Chapter 2

Controlled computations from Lattice-based Cryptography

2.1 Lattice-based cryptography

Lattice-based cryptography has its origins in the seminal work of Ajtai [9] who constructed one-way functions based on the Short Integer Solutions (SIS) problem and proved that solving the SIS problem "on average" i.e. for a randomly chosen sample is at least as hard as finding approximate solutions to the "worst-case" instance of a well-studied computational problem on lattices. Constant or even sub-polynomial ($n^{o(1)}$) approximations to these lattice problems are NP-hard [10, 161, 143, 125]. The cryptography constructions rely on the hardness of these problems with polynomial approximations factors. Even though these are not proved to be NP-hard, in fact not expected to NP-hard, no efficient algorithms are known even with sub-exponential approximation factors.

Modern lattice-based cryptography started with another seminal work by Regev [186] who introduced the problem of Learning With Errors (LWE) and proved a similar worst-case to average-case reduction from the lattice problems. The LWE assumption has been shown to be extremely powerful with it forming the base for the first instantiations of powerful cryptographic constructions like fully-homomorphic encryption [95] and reusable-garbled circuits [107, 104]. Constructions based on variants of LWE and SIS are also one of the leading contenders in the NIST standardization process for quantum-safe key-exchange and public-key signature schemes [171].

This chapter explains our construction of attribute-based encryption scheme from the LWE assumption for the computation class of branching programs. Our construction

provides short keys and polynomial approximation factors in the security reductions from the underlying lattice problems. This would in turn lead to constructions for single-key functional encryption with similar security and efficiency guarantees [104].

2.2 Attribute-based encryption

Attribute-Based Encryption (ABE) was introduced by Sahai and Waters [189] in order to realize the vision of fine-grained access control to encrypted data. Using ABE, a user can encrypt a message μ with respect to a public attribute-vector $\mathbf{x}$ to obtain a ciphertext $\mathsf{ct}_{\mathbf{x}}$. Anyone holding a secret key sk_P, associated with an access policy P, can decrypt the message μ if $P(\mathbf{x}) = 1$. Moreover, the security notion guarantees that no collusion of adversaries holding secret keys $\mathsf{sk}_{P_1}, \ldots, \mathsf{sk}_{P_t}$ can learn anything about the message μ if none of the individual keys allow to decrypt it. Until recently, candidate constructions of ABE were limited to restricted classes of access policies that test for equality (IBE), boolean formulas and inner-products: [73, 36, 113, 205, 149, 148, 5, 61, 6, 49].

In recent breakthroughs Gorbunov, Vaikuntanathan and Wee [107] and Garg, Gentry, Halevi, Sahai and Waters [94] constructed ABE schemes for arbitrary boolean predicates. The GVW construction is based on the LWE problem with sub-exponential approximation factors, whereas GGHSW relies on hardness of a (currently) stronger assumption over the multilinear map candidates [92, 74, 96]. But in both these ABE schemes, the size of the secret keys had a multiplicative dependence on the size of the predicate: $|P| \cdot \mathrm{poly}(\lambda, d)$ (where d is the depth of the circuit representation of the predicate). In a subsequent work, Boneh et al. [42] showed how to construct ABE for arithmetic predicates with short secret keys: $|P| + \mathrm{poly}(\lambda, d)$, also assuming hardness of LWE with sub-exponential approximation factors. However, in [107], the authors also showed an additional construction for a family of branching programs under a milder and quantitatively better assumption: hardness of LWE with polynomial approximation factors. Basing the security on LWE with polynomial approximation factors results in two main advantages. First, the security of the resulting construction relies on the hardness of a much milder LWE assumption. But moreover, the resulting instantiation has better parameters – with a small modulus q – leading directly to practical efficiency improvements.

In this work, we focus on constructing an ABE scheme under milder security assumptions and better performance guarantees. We concentrate on ABE for a family of branching programs which is sufficient for most existing applications such as medical and multimedia data sharing [13, 150, 176].

First, we summarize the two most efficient results from learning with errors problem translated to the setting of branching programs (via standard Barrington's theorem [30]). Let L be the length of a branching program P and let λ denote the security parameter. Then,

- [107]: There exists an ABE scheme for length L branching programs with *large secret keys* based on the security of *LWE with polynomial approximation factors*. In particular, in the instantiation $|\mathsf{sk}_P| = |L| \times \mathrm{poly}(\lambda, \log L)$ and $q = \mathrm{poly}(L, \lambda)$.
- [42]: There exists an ABE scheme for length L branching programs with *small secret keys* based on the security of *LWE with quasi-polynomial approximation factors*. In particular, $|\mathsf{sk}_P| = |L| + \mathrm{poly}(\lambda, \log L)$, $q = \mathrm{poly}(\lambda)^{\log L}$.

To advance the state of the art for both theoretical and practical reasons, the natural question that arises is whether we can obtain the best of both worlds and:

> *Construct an ABE for branching programs with small secret keys based on the security of LWE with polynomial approximation factors?*

2.2.1 Our Results

We present a new efficient construction of ABE for branching programs from a mild LWE assumption. Our result can be summarized in the following theorem.

Theorem 2.2.1 (informal)**.** *There exists a selectively-secure Attribute-Based Encryption for a family of length-L branching programs with* small secret keys *based on the security of* LWE with polynomial approximation factors*. More formally, the size of the secret key* sk_P *is* $L + \mathrm{poly}(\lambda, \log L)$ *and modulo* $q = \mathrm{poly}(L, \lambda)$*, where* λ *is the security parameter.*

Furthermore, we can extend our construction to support arbitrary length branching programs by setting q to some small super-polynomial.

As an additional contribution, our techniques lead to a new efficient constructing of homomorphic signatures for branching programs. In particular, Gorbunov et al. [109] showed how to construct homomorphic signatures for circuits based on the simulation techniques of Boneh et al. [42] in the context of ABE. Their resulting construction is secure based on the short integer solution (SIS) problem with *sub-exponential approximation factors* (or quasi-polynomial in the setting of branching programs). Analogously, our simulation algorithm presented in Section 2.4.4 can be used directly to construct homomorphic signatures for branching programs based on SIS with *polynomial approximation factors*.

Theorem 2.2.2 (informal)**.** *There exists a homomorphic signatures scheme for the family of length-L branching programs based on the security of* SIS with polynomial approximation factors.

High Level Overview. The starting point of our ABE construction is the ABE scheme for circuits with short secret keys by Boneh et al. [42]. At the heart of their construction is a fully key-homomorphic encoding scheme.

It encodes $a \in \{0,1\}$ with respect to a public key $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n\times m}$ in a "noisy" sample:

$$\psi_{\mathbf{A},a} = (\mathbf{A} + a \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}$$

where $\mathbf{s} \xleftarrow{\$} \mathbb{Z}_q^n$ and $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$ are fixed across all the encodings and $\mathbf{e} \xleftarrow{\$} \chi^m$ (for some noise distribution χ) is chosen independently every time. The authors show that one can turn such a key-homomorphic encoding scheme, where homomorphism is satisfied over the encoded values and over the public keys simultaneously, into an attribute based encryption scheme for circuits.

Our first key observation is the asymmetric noise growth in their homomorphic multiplication over the encodings. Consider ψ_1, ψ_2 to be the encodings of a_1, a_2 under public keys $\mathbf{A}_1, \mathbf{A}_2$. To achieve multiplicative homomorphism, their first step is to achieve homomorphism over a_1 and a_2 by computing

$$a_1 \cdot \psi_2 = (a_1 \cdot \mathbf{A}_2 + (a_1 a_2) \cdot \mathbf{G})^{\intercal}\, \mathbf{s} + a_1 \mathbf{e}_2 \tag{2.1}$$

Now, since homomorphism over the public key matrices must also be satisfied in the resulting encoding *independently* of a_1, a_2 we must replace $a_1 \cdot \mathbf{A}_2$ in Equation 2.1 with operations over $\mathbf{A}_1, \mathbf{A}_2$ only. To do this, we can use the first encoding $\psi_1 = (\mathbf{A}_1 + a_1 \cdot \mathbf{G})^{\intercal} + \mathbf{e}_1$ and replace $a_1 \cdot \mathbf{G}$ with $a_1 \cdot \mathbf{A}_2$ as follows. First, compute $\widetilde{\mathbf{A}}_2 \in \{0,1\}^{m\times m}$ such that $\mathbf{G} \cdot \widetilde{\mathbf{A}}_2 = \mathbf{A}_2$. (Finding such $\widetilde{\mathbf{A}}_2$ is possible since the "trapdoor" of $\mathbf{G}$ is known publicly). Then compute

$$\begin{aligned}
(\widetilde{\mathbf{A}}_2)^{\intercal} \cdot \psi_1 &= \widetilde{\mathbf{A}}_2^{\intercal} \cdot ((\mathbf{A}_1 + a_1 \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}_1) \\
&= \left(\mathbf{A}_1\widetilde{\mathbf{A}}_2 + a_1 \cdot \mathbf{G}\widetilde{\mathbf{A}}_2\right)^{\intercal} \mathbf{s} + \widetilde{\mathbf{A}}_2\mathbf{e}_1 \\
&= \left(\mathbf{A}_1\widetilde{\mathbf{A}}_2 + a_1 \cdot \mathbf{A}_2\right)^{\intercal} \mathbf{s} + \mathbf{e}_1' \qquad (2.2)
\end{aligned}$$

Subtracting Equation 2.2 from Equation 2.1, we get $\left(-\mathbf{A}_1\widetilde{\mathbf{A}}_2 + (a_1 a_2) \cdot \mathbf{G}\right)^{\intercal} \mathbf{s} + \mathbf{e}'$ which is an encoding of $a_1 a_2$ under the public key $\mathbf{A}^{\times} := -\mathbf{A}_1\widetilde{\mathbf{A}}_2$. Thus,

$$\psi_{\mathbf{A}^{\times},a^{\times}} := a_1 \cdot \psi_2 - \widetilde{\mathbf{A}}_2^{\intercal} \cdot \psi_1$$

where $a^\times := a_1 a_2$. Here, $\mathbf{e}'$ remains small enough because $\widetilde{\mathbf{A}}_2$ has small (binary) entries. We observe that the new noise $\mathbf{e}' = a_1\mathbf{e}_2 - \widetilde{\mathbf{A}}_2\mathbf{e}_1$ grows asymmetrically. That is, the poly(n) multiplicative increase always occurs with respect to the first noise $\mathbf{e}_1$. Naïvely evaluating k levels of multiplicative homomorphism results in a noise of magnitude poly$(n)^k$. Can we manage the noise growth by some careful design of the order of homomorphic operations?

To achieve this, comes our second idea: design evaluation algorithms for a "sequential" representation of a matrix branching program to carefully manage the noise growth following the Brakerski-Vaikuntanathan paradigm in the context of fully-homomorphic encryption [51].

First, to generate a ciphertext with respect to an attribute vector $\mathbf{x} = (x_1, \ldots, x_\ell)$ we publish encodings of its individual bits:

$$\psi_i \approx (\mathbf{A}_i + x_i \cdot \mathbf{G})^\intercal \mathbf{s}$$

We also publish encoding of an initial start state 0:[1]

$$\psi_0^v \approx (\mathbf{A}_0^v + v_0 \cdot \mathbf{G})^\intercal \mathbf{s}$$

The message μ is encrypted under encoding $\mathbf{u}^\intercal\mathbf{s} + e$ (where $\mathbf{u}$ is treated as the public key) and during decryption the user should obtain a value $\approx \mathbf{u}^\intercal\mathbf{s}$ from $\{\psi_i\}_{i\in[\ell]}, \psi_0^v$ iff $P(\mathbf{x}) = 1$.

Now, suppose the user wants to evaluate a branching program P on the attribute vector $\mathbf{x}$. Informally, the evaluation of a branching program proceeds in steps updating a special state vector. The next state is determined by the current state and one of the input bits (pertaining to this step). Viewing the sequential representation of the branching program allows us to update the state using only a *single* multiplication and a few additions. Suppose v_t represents the state of the program P at step t and the user holds its corresponding encoding ψ_t^v (under some public key). To obtain ψ_{t+1}^v the user needs to use ψ_i (for some i determined by the program). Leveraging on the asymmetry, the state can be updated by multiplying ψ_i with the matrix $\widetilde{\mathbf{A}}_t^v$ corresponding to the encoding ψ_t^v (and then following a few simple addition steps). Since ψ_i always contains a "fresh" noise (which is never increased as we progress evaluating the program), the noise in ψ_{t+1}^v increases from the noise in ψ_t^v only by a *constant additive* factor! As a result, after k steps in the evaluation procedure the noise will be bounded by $k \cdot \text{poly}(n)$. Eventually, if $P(\mathbf{x}) = 1$, the user will learn $\approx \mathbf{u}^\intercal\mathbf{s}$ and be able to recover μ (we refer the reader to the main construction for details).

[1] Technically, we need to publish encodings of 5 states, but we simplify the notation in the introduction for conceptual clarify.

The main challenge in "riding on asymmetry" for attribute-based encryption is the requirement for satisfying parallel homomorphic properties: we must design separate homomorphic algorithms for operating over the public key matrices and over the encodings that allow for correct decryption. First, we define and design an algorithm for public key homomorphic operations that works specially for branching programs. Second, we design a homomorphic algorithm that works over the encodings that *preserves the homomorphism over public key matrices and the bits*[2] and *carefully manages the noise growth* as illustrated above. To prove the security, we need to argue that no collusion of users is able to learn anything about the message given many secret keys for programs that do not allow for decryption individually. We design a separate public-key simulation algorithm to accomplish this.

2.2.2 Applications

We summarize some of the known applications of attribute-based encryption. Parno, Raykova and Vaikuntanathan [178] showed how to use ABE to design (publicly) verifiable two-message delegation delegation scheme with a pre-processing phase. Also as we mentioned, Goldwasser, Kalai, Popa, Vaikuntanathan and Zeldovich [104] showed how to use ABE as a critical building block to construct succinct one-query functional encryption, reusable garbled circuits, token-based obfuscation and homomorphic encryption for Turing machines. Our efficiency improvements for branching programs can be carried into all these applications.

2.2.3 Other Related Work

A number of works optimized attribute-based encryption for boolean formulas: Attrapadung et al. [22] and Emura et al. [84] designed ABE schemes with constant size ciphertext from bilinear assumptions. For arbitrary circuits, Boneh et al. [42] also showed an ABE with constant size ciphertext from multilinear assumptions. ABE can also be viewed as a special case of functional encryptions [37]. Gorbunov et al. [106] showed functional encryption for arbitrary functions in a bounded collusion model from standard public-key encryption scheme. Garg et al. [93] presented a functional encryption for unbounded collusions for arbitrary functions under a weaker security model from multilinear assumptions. More recently, Gorbunov et al. exploited the asymmetry in

[2]These bits represent the bits of the attribute vector in the ABE scheme

the noise growth in [42] in a different context of design of a predicate encryption scheme based on standard LWE [108].

2.2.4 Extensions

We note a few possible extensions on our basic construction that lead to further efficiency improvements. First, we can support arbitrary width branching programs by appropriately increasing the dimension of the state vector in the encryption. Second, we can switch to an arithmetic setting, similarly as it was done in [42].

2.2.5 Organization of this chapter

In Section 2.3 we present the lattice preliminaries, definitions for ABE and branching programs. In Section 2.4 we present our main evaluation algorithms and build our ABE scheme in Section 2.5. We present a concrete instantiation of the parameters in Section 2.6.

2.3 Preliminaries

Notation. Let PPT denote probabilistic polynomial-time. For any integer $q \geq 2$, we let $\mathbb{Z}_q$ denote the ring of integers modulo q and we represent $\mathbb{Z}_q$ as integers in $(-q/2, q/2]$. We let $\mathbb{Z}_q^{n \times m}$ denote the set of $n \times m$ matrices with entries in $\mathbb{Z}_q$. We use bold capital letters (e.g. $\mathbf{A}$) to denote matrices, bold lowercase letters (e.g. $\mathbf{x}$) to denote vectors. The notation $\mathbf{A}^\intercal$ denotes the transpose of the matrix $\mathbf{A}$. If $\mathbf{A}_1$ is an $n \times m$ matrix and $\mathbf{A}_2$ is an $n \times m'$ matrix, then $[\mathbf{A}_1 \| \mathbf{A}_2]$ denotes the $n \times (m + m')$ matrix formed by concatenating $\mathbf{A}_1$ and $\mathbf{A}_2$. A similar notation applies to vectors. When doing matrix-vector multiplication we always view vectors as column vectors. Also, $[n]$ denotes the set of numbers $1, \ldots, n$.

We say a function $f(n)$ is *negligible* if it is $O(n^{-c})$ for all $c > 0$, and we use $\text{negl}(n)$ to denote a negligible function of n. We say $f(n)$ is *polynomial* if it is $O(n^c)$ for some $c > 0$, and we use $\text{poly}(n)$ to denote a polynomial function of n. We say an event occurs with *overwhelming probability* if its probability is $1 - \text{negl}(n)$. The function $\log x$ is the base 2 logarithm of x. The notation $\lfloor x \rceil$ denotes the nearest integer to x, rounding towards 0 for half-integers.

2.3.1 Lattice Preliminaries

Learning With Errors (LWE) Assumption

The LWE problem was introduced by Regev [186], who showed that solving it *on the average* is as hard as (quantumly) solving several standard lattice problems *in the worst case.*

Definition 2.3.1 (LWE)**.** *For an integer* $q = q(n) \geq 2$ *and an error distribution* $\chi = \chi(n)$ *over* $\mathbb{Z}_q$*, the learning with errors problem* $\mathsf{dLWE}_{n,m,q,\chi}$ *is to distinguish between the following pairs of distributions:*

$$\{\mathbf{A}, \mathbf{A}^{\intercal}\mathbf{s} + \mathbf{x}\} \quad \textit{and} \quad \{\mathbf{A}, \mathbf{u}\}$$

where $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}, \mathbf{s} \xleftarrow{\$} \mathbb{Z}_q^n, \mathbf{x} \xleftarrow{\$} \chi^m, \mathbf{u} \xleftarrow{\$} \mathbb{Z}_q^m$.

Connection to lattices. Let $B = B(n) \in \mathbb{N}$. A family of distributions $\chi = \{\chi_n\}_{n \in \mathbb{N}}$ is called B-bounded if

$$\Pr[\chi \in \{-B, \ldots, B-1, B\}] = 1.$$

There are known quantum [186] and classical [180] reductions between $\mathsf{dLWE}_{n,m,q,\chi}$ and approximating short vector problems in lattices in the worst case, where χ is a B-bounded (truncated) discretized Gaussian for some appropriate B. The state-of-the-art algorithms for these lattice problems run in time nearly exponential in the dimension n [12, 163]; more generally, we can get a 2^k-approximation in time $2^{\tilde{O}(n/k)}$. Throughout this paper, the parameter $m = \mathrm{poly}(n)$, in which case we will shorten the notation slightly to $\mathsf{LWE}_{n,q,\chi}$.

Trapdoors for Lattices and LWE

Gaussian distributions. Let $D_{\mathbb{Z}^m,\sigma}$ be the truncated discrete Gaussian distribution over $\mathbb{Z}^m$ with parameter σ, that is, we replace the output by $\mathbf{0}$ whenever the $|| \cdot ||_\infty$ norm exceeds $\sqrt{m} \cdot \sigma$. Note that $D_{\mathbb{Z}^m,\sigma}$ is $\sqrt{m} \cdot \sigma$-bounded.

Lemma 2.3.1 (Lattice Trapdoors [11, 97, 162])**.** *There is an efficient randomized algorithm* $\mathsf{TrapSamp}(1^n, 1^m, q)$ *that, given any integers* $n \geq 1$*,* $q \geq 2$*, and sufficiently large* $m = \Omega(n \log q)$*, outputs a parity check matrix* $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ *and a 'trapdoor' matrix* $\mathbf{T_A} \in \mathbb{Z}^{m \times m}$ *such that the distribution of* $\mathbf{A}$ *is* $\mathrm{negl}(n)$*-close to uniform.*

Moreover, there is an efficient algorithm $\mathsf{SampleD}$ *that with overwhelming probability over all random choices, does the following: For any* $\mathbf{u} \in \mathbb{Z}_q^n$*, and large enough* $s =$

$\Omega(\sqrt{n \log q})$, *the randomized algorithm* $\mathsf{SampleD}(\mathbf{A}, \mathbf{T_A}, \mathbf{u}, s)$ *outputs a vector* $\mathbf{r} \in \mathbb{Z}^m$ *with norm* $||\mathbf{r}||_\infty \leq ||\mathbf{r}||_2 \leq s\sqrt{n}$ *(with probability 1). Furthermore, the following distributions of the tuple* $(\mathbf{A}, \mathbf{T_A}, \mathbf{U}, \mathbf{R})$ *are within* $\text{negl}(n)$ *statistical distance of each other for any polynomial* $k \in \mathbb{N}$*:*

- $(\mathbf{A}, \mathbf{T_A}) \leftarrow \mathsf{TrapSamp}(1^n, 1^m, q)$*;* $\mathbf{U} \leftarrow \mathbb{Z}_q^{n \times k}$*;* $\mathbf{R} \leftarrow \mathsf{SampleD}(\mathbf{A}, \mathbf{T_A}, \mathbf{U}, s)$.
- $(\mathbf{A}, \mathbf{T_A}) \leftarrow \mathsf{TrapSamp}(1^n, 1^m, q)$*;* $\mathbf{R} \leftarrow (D_{\mathbb{Z}^m, s})^k$*;* $\mathbf{U} := \mathbf{AR} \pmod{q}$.

Sampling algorithms

We will use the following algorithms to sample short vectors from specific lattices. Looking ahead, the algorithm $\mathsf{SampleLeft}$ [5, 61] will be used to sample keys in the real system, while the algorithm $\mathsf{SampleRight}$ [5] will be used to sample keys in the simulation.

Algorithm $\mathsf{SampleLeft}(\mathbf{A}, \mathbf{B}, \mathbf{T_A}, \mathbf{u}, \alpha)$**:**

Inputs: a full rank matrix $\mathbf{A}$ in $\mathbb{Z}_q^{n \times m}$, a "short" basis $\mathbf{T}_A$ of $\Lambda_q^\perp(\mathbf{A})$, a matrix $\mathbf{B}$ in $\mathbb{Z}_q^{n \times m_1}$, a vector $\mathbf{u} \in \mathbb{Z}_q^n$, and a Gaussian parameter α. (2.3)

Output: Let $\mathbf{F} := (\mathbf{A} \parallel \mathbf{B})$. The algorithm outputs a vector $\mathbf{e} \in \mathbb{Z}^{m+m_1}$ in the coset $\Lambda_{\mathbf{F}+\mathbf{u}}$. (2.4)

Theorem 2.3.2 ([5, Theorem 17], [61, Lemma 3.2])**.** *Let* $q > 2$*,* $m > n$ *and* $\alpha > \|\mathbf{T_A}\|_{\mathsf{GS}} \cdot \omega(\sqrt{\log(m + m_1)})$*. Then* $\mathsf{SampleLeft}(\mathbf{A}, \mathbf{B}, \mathbf{T_A}, \mathbf{u}, \alpha)$ *taking inputs as in Equation 2.3 outputs a vector* $\mathbf{e} \in \mathbb{Z}^{m+m_1}$ *distributed statistically close to* $D_{\Lambda_{\mathbf{F}+\mathbf{u}}, \alpha}$*, where* $\mathbf{F} := (\mathbf{A} \parallel \mathbf{B})$.

where $\|\mathbf{T}\|_{\mathsf{GS}}$ refers to the norm of Gram-Schmidt orthogonalisation of $\mathbf{T}$. We refer the readers to [5] for more details.

Algorithm $\mathsf{SampleRight}(\mathbf{A}, \mathbf{G}, \mathbf{R}, \mathbf{T_G}, \mathbf{u}, \alpha)$**:**

Inputs: matrices $\mathbf{A}$ in $\mathbb{Z}_q^{n \times k}$ and $\mathbf{R}$ in $\mathbb{Z}^{k \times m}$, a full rank matrix $\mathbf{G}$ in $\mathbb{Z}_q^{n \times m}$, a "short" basis $\mathbf{T_G}$ of $\Lambda_q^\perp(\mathbf{G})$, a vector $\mathbf{u} \in \mathbb{Z}_q^n$, and a Gaussian parameter α. (2.5)

Output: Let $\mathbf{F} := (\mathbf{A} \parallel \mathbf{AR} + \mathbf{G})$. The algorithm outputs a vector $\mathbf{e} \in \mathbb{Z}^{m+k}$ in the coset $\Lambda_{\mathbf{F}+\mathbf{u}}$. (2.6)

Often the matrix $\mathbf{R}$ given to the algorithm as input will be a random matrix in $\{1,-1\}^{m\times m}$. Let S^m be the m-sphere $\{\mathbf{x} \in \mathbb{R}^{m+1} \;:\; \|\mathbf{x}\| = 1\}$. We define $s_R := \|\mathbf{R}\| := \sup_{\mathbf{x}\in S^{m-1}} \|\mathbf{R}\cdot\mathbf{x}\|$.

Theorem 2.3.3 ([5, Theorem 19]). *Let $q > 2, m > n$ and $\alpha > \|\mathbf{T_G}\|_{\mathsf{GS}} \cdot s_R \cdot \omega(\sqrt{\log m})$. Then $\mathsf{SampleRight}(\mathbf{A}, \mathbf{G}, \mathbf{R}, \mathbf{T_G}, \mathbf{u}, \alpha)$ taking inputs as in Equation 2.5 outputs a vector $\mathbf{e} \in \mathbb{Z}^{m+k}$ distributed statistically close to $D_{\Lambda_{\mathbf{F}+\mathbf{u}},\alpha}$, where $\mathbf{F} := (\mathbf{A} \parallel \mathbf{AR} + \mathbf{G})$.*

Primitive matrix

We use the primitive matrix $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$ defined in [162]. This matrix has a trapdoor $\mathbf{T_G}$ such that $\|\mathbf{T_G}\|_\infty = 2$.

We also define an algorithm $\mathsf{invG} : \mathbb{Z}_q^{n\times m} \to \mathbb{Z}_q^{m\times m}$ which deterministically derives a pre-image $\widetilde{\mathbf{A}}$ satisfying $\mathbf{G}\cdot\widetilde{\mathbf{A}} = \mathbf{A}$. From [162], there exists a way to get $\widetilde{\mathbf{A}}$ such that $\widetilde{\mathbf{A}} \in \{0,1\}^{m\times m}$.

2.3.2 Attribute-Based Encryption definition

An attribute-based encryption scheme $\mathcal{ABE}$ [113] for a class of predicates[3] $\mathcal{P}$ with ℓ bit inputs and message space $\mathcal{M}$ consists of a tuple of p.p.t. algorithms $(\mathsf{Params}, \mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$:

$\mathsf{Params}(1^\lambda) \to \mathsf{pp}$: The parameter generation algorithm takes the security parameter 1^λ and outputs a public parameter pp which is implicitly given to all the other algorithms of the scheme.

$\mathsf{Setup}(1^\ell) \to (\mathsf{mpk}, \mathsf{msk})$: The setup algorithm gets as input the length ℓ of the input index, and outputs the master public key mpk, and the master key msk.

$\mathsf{Enc}(\mathsf{mpk}, x, \mu) \to \mathsf{ct}_\mathbf{x}$: The encryption algorithm gets as input mpk, an index $x \in \{0,1\}^\ell$ and a message $\mu \in \mathcal{M}$. It outputs a ciphertext $\mathsf{ct}_\mathbf{x}$.

$\mathsf{KeyGen}(\mathsf{msk}, P) \to \mathsf{sk}_P$: The key generation algorithm gets as input msk and a predicate specified by $P \in \mathcal{P}$. It outputs a secret key sk_P.

$\mathsf{Dec}(\mathsf{ct}_\mathbf{x}, \mathsf{sk}_P) \to \mu$: The decryption algorithm gets as input $\mathsf{ct}_\mathbf{x}$ and sk_P, and outputs either $\perp$ or a message $\mu \in \mathcal{M}$.

[3] Here, a predicate is any program or circuit that outputs a single bit.

Definition 2.3.2 (Correctness)**.** *We require that for all* $(\mathbf{x}, P)$ *such that* $P(\mathbf{x}) = 1$ *and for all* $\mu \in \mathcal{M}$*, we have* $\Pr[\mathsf{ct}_{\mathbf{x}} \leftarrow \mathsf{Enc}(\mathsf{mpk}, \mathbf{x}, \mu); \mathsf{Dec}(\mathsf{ct}_{\mathbf{x}}, \mathsf{sk}_P) = \mu)] = 1$ *where the probability is taken over* $\mathsf{pp} \leftarrow \mathsf{Params}(1^\lambda)$, $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\ell)$ *and the coins of all the algorithms in the expression above.*

Definition 2.3.3 (Security)**.** *For a stateful adversary* $\mathcal{A}$*, we define the advantage function* $\mathsf{Adv}^{\mathrm{ABE}}_{\mathcal{A}}(\lambda)$ *to be*

$$\Pr\left[b = b' : \begin{array}{l} \mathbf{x}^* \leftarrow \mathcal{A}(1^\lambda, 1^\ell); \\ \mathsf{pp} \leftarrow \mathsf{Params}(1^\lambda); \\ (\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\ell, \mathbf{x}^*); \\ (\mu_0, \mu_1) \leftarrow \mathcal{A}^{\mathsf{Keygen}(\mathsf{msk}, \cdot)}(\mathsf{mpk}), |\mu_0| = |\mu_1|; \\ b \xleftarrow{\$} \{0,1\}; \\ \mathsf{ct}_{\mathbf{x}} \leftarrow \mathsf{Enc}(\mathsf{mpk}, \mathbf{x}, \mu_b); \\ b' \leftarrow \mathcal{A}^{\mathsf{Keygen}(\mathsf{msk}, \cdot)}(\mathsf{ct}_{\mathbf{x}}) \end{array} \right] - \frac{1}{2}$$

with the restriction that all queries y *that* $\mathcal{A}$ *makes to* $\mathsf{Keygen}(\mathsf{msk}, \cdot)$ *satisfies* $P(\mathbf{x}^*) = 0$ *(that is,* sk_P *does not decrypt* $\mathsf{ct}_{\mathbf{x}}$*). An attribute-based encryption scheme is* selectively secure *if for all PPT adversaries* $\mathcal{A}$*, the advantage* $\mathsf{Adv}^{\mathrm{ABE}}_{\mathcal{A}}(\lambda)$ *is a negligible function in* λ*.*

2.3.3 Branching Programs

We define branching programs similar to [51]. A width-w branching program BP of length L with input space $\{0,1\}^\ell$ and s states (represented by $[s]$) is a sequence of L tuples of the form $(\mathsf{var}(t), \sigma_{t,0}, \sigma_{t,1})$ where

- $\sigma_{t,0}$ and $\sigma_{t,1}$ are injective functions from $[s]$ to itself.
- $\mathsf{var} : [L] \to [\ell]$ is a function that associates the t-th tuple $\sigma_{t,0}, \sigma_{t,1}$ with the input bit $x_{\mathsf{var}(t)}$.

The branching program BP on input $\mathbf{x} = (x_1, \ldots, x_\ell)$ computes its output as follows. At step t, we denote the state of the computation by $\eta_t \in [s]$. The initial state is $\eta_0 = 1$. In general, η_t can be computed recursively as

$$\eta_t = \sigma_{t, x_{\mathsf{var}(t)}}(\eta_{t-1})$$

Finally, after L steps, the output of the computation $\mathsf{BP}(\mathbf{x}) = 1$ if $\eta_L = 1$ and 0 otherwise.

As done in [51], we represent states with bits rather than numbers to bound the noise growth. In particular, we represent the state $\eta_t \in [s]$ by a unit vector $\mathbf{v}_t \in \{0,1\}^s$. The idea is that $\mathbf{v}_t[i] = 1$ if and only if $\sigma_{t,x_{\mathsf{var}(t)}}(\eta_{t-1}) = i$. Note that we can also write the above expression as $\mathbf{v}_t[i] = 1$ if and only if either:

- $\mathbf{v}_{t-1}\left[\sigma_{t,0}^{-1}(i)\right] = 1$ and $x_{\mathsf{var}(t)} = 0$
- $\mathbf{v}_{t-1}\left[\sigma_{t,1}^{-1}(i)\right] = 1$ and $x_{\mathsf{var}(t)} = 1$

This latter form will be useful for us since it can be captured by the following formula. For $t \in [L]$ and $i \in [s]$,

$$\begin{aligned}\mathbf{v}_t[i] &:= \mathbf{v}_{t-1}\left[\sigma_{t,0}^{-1}(i)\right] \cdot (1 - x_{\mathsf{var}(t)}) + \mathbf{v}_{t-1}\left[\sigma_{t,1}^{-1}(i)\right] \cdot x_{\mathsf{var}(t)} \\ &= \mathbf{v}_{t-1}\left[\gamma_{t,i,0}\right] \cdot (1 - x_{\mathsf{var}(t)}) + \mathbf{v}_{t-1}\left[\gamma_{t,i,1}\right] \cdot x_{\mathsf{var}(t)}\end{aligned}$$

where $\gamma_{t,i,0} := \sigma_{t,0}^{-1}(i)$ and $\gamma_{t,i,1} = \sigma_{t,1}^{-1}(i)$ can be publicly computed from the description of the branching program. Hence, $\left\{\mathsf{var}(t), \{\gamma_{t,i,0}, \gamma_{t,i,1}\}_{i\in[s]}\right\}_{t\in[L]}$ is also valid representation of a branching program BP.

For clarity of presentation, we will deal with width-5 permutation branching programs, which is shown to be equivalent to the circuit class $\mathcal{NC}^1$ [30]. Hence, we have $s = w = 5$ and the functions σ_0, σ_1 are permutations on [5].

2.4 Our Evaluation Algorithms

In this section we describe the key evaluation and encoding (ciphertext) evaluation algorithms that will be used in our ABE construction. The algorithms are carefully designed to manage the noise growth in the LWE encodings *and* to preserve parallel homomorphism over the public keys and the encoded values.

2.4.1 Basic Homomorphic Operations

We first describe basic homomorphic addition and multiplication algorithms over the public keys and encodings (ciphertexts) based on the techniques developed by Boneh et al. [42].

Definition 2.4.1 (LWE Encoding)**.** *For any matrix* $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n\times m}$*, we define an LWE encoding of a bit* $a \in \{0,1\}$ *with respect to a (public) key* $\mathbf{A}$ *and randomness* $\mathbf{s} \xleftarrow{\$} \mathbb{Z}_q^n$ *as*

$$\psi_{\mathbf{A},\mathbf{s},a} = (\mathbf{A} + a\cdot\mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e} \in \mathbb{Z}_q^m$$

for error vector $\mathbf{e} \xleftarrow{\$} \chi^m$ *and an (extended) primitive matrix* $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$*.*

In our construction, however, all encodings will be under the same LWE secret $\mathbf{s}$, hence for simplicity we will simply refer to such an encoding as $\psi_{\mathbf{A},a}$.

Definition 2.4.2 (Noise Function)**.** *For every* $\mathbf{A} \in \mathbb{Z}_q^{n\times m}, \mathbf{s} \in \mathbb{Z}_q^n$ *and encoding* $\psi_{\mathbf{A},a} \in \mathbb{Z}_q^m$ *of a bit* $a \in \{0,1\}$ *we define a noise function as*

$$\mathsf{Noise}_{\mathbf{s}}(\psi_{\mathbf{A},a}) := ||\psi_{\mathbf{A},a} - (\mathbf{A} + a\cdot\mathbf{G})^{\intercal}\mathbf{s} \mod q||_{\infty}$$

Looking ahead, in Lemma 2.5.1 we show that if the noise obtained after applying homomorphic evaluation is $\leq q/4$, then our ABE scheme will decrypt the message correctly. Now we define the basic additive and multiplicative operations on the encodings of this form, as per [42]. In their context, they refer to a matrix $\mathbf{A}$ as the "public key" and $\psi_{\mathbf{A},a}$ as a ciphertext.

Homomorphic addition

This algorithm takes as input two encodings $\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'}$ and outputs the sum of them. Let $\mathbf{A}^+ = \mathbf{A} + \mathbf{A}'$ and $a^+ = a + a'$.

$$\mathsf{Add}_{\mathsf{en}}(\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'}) : \text{ Output } \psi_{\mathbf{A}^+,a^+} := \psi_{\mathbf{A},a} + \psi_{\mathbf{A}',a'} \mod q$$

Lemma 2.4.1 (Noise Growth in $\mathsf{Add}_{\mathsf{en}}$)**.** *For any two valid encodings* $\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'} \in \mathbb{Z}_q^m$*, let* $\mathbf{A}^+ = \mathbf{A} + \mathbf{A}'$ *and* $a^+ = a + a'$ *and* $\psi_{\mathbf{A}^+,a^+} = \mathsf{Add}_{\mathsf{en}}(\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'})$*, then we have*

$$\mathsf{Noise}_{\mathbf{A}^+,a^+}(\psi_{\mathbf{A}^+,a^+}) \leq \mathsf{Noise}_{\mathbf{A},a}(\psi_{\mathbf{A},a}) + \mathsf{Noise}_{\mathbf{A}',a'}(\psi_{\mathbf{A}',a'})$$

Proof. Given two encodings we have,

$$\begin{aligned}
\psi_{\mathbf{A}^+,a^+} &= \psi_{\mathbf{A},a} + \psi_{\mathbf{A}',a'} \\
&= ((\mathbf{A} + a\cdot\mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}) + ((\mathbf{A}' + a'\cdot\mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}') \\
&= ((\mathbf{A} + \mathbf{A}') + (a + a')\cdot\mathbf{G})^{\intercal}\,\mathbf{s} + (\mathbf{e} + \mathbf{e}') \\
&= (\mathbf{A}^+ + a^+\cdot\mathbf{G})^{\intercal}\mathbf{s} + (\mathbf{e} + \mathbf{e}')
\end{aligned}$$

Thus, from the definition of the noise function, it follows that

$$\mathsf{Noise}_{\mathbf{A}^+,a^+}(\psi_{\mathbf{A},a} + \psi_{\mathbf{A}',a'}) \leq \mathsf{Noise}_{\mathbf{A},a}(\psi_{\mathbf{A},a}) + \mathsf{Noise}_{\mathbf{A}',a'}(\psi_{\mathbf{A}',a'})$$

□

Homomorphic multiplication

This algorithm takes in two encodings $\psi_{\mathbf{A},a} = (\mathbf{A} + a \cdot \mathbf{G})^\mathsf{T}\mathbf{s} + \mathbf{e}_1$ and $\psi_{\mathbf{A}',a'} = (\mathbf{A}' + a' \cdot \mathbf{G})^\mathsf{T}\mathbf{s} + \mathbf{e}_2$ and outputs an encoding $\psi_{\mathbf{A}^\times,a^\times}$ where $\mathbf{A}^\times = -\mathbf{A}\widetilde{\mathbf{A}'}$ and $a^\times = aa'$ as follows:

$$\mathsf{Multiply}_{\mathsf{en}}(\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'}) : \text{ Output } \psi_{\mathbf{A}^\times,a^\times} := -\widetilde{\mathbf{A}'}^\mathsf{T} \cdot \psi + a \cdot \psi'.$$

Note that this process requires the knowledge of the attribute a in clear.

Lemma 2.4.2 (Noise Growth in $\mathsf{Multiply}_{\mathsf{en}}$)**.** *For any two valid encodings $\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'} \in \mathbb{Z}_q^m$, let $\mathbf{A}^\times = -\mathbf{A}\widetilde{\mathbf{A}'}$ and $a^\times = aa'$ and $\psi_{\mathbf{A}^\times,a^\times} = \mathsf{Multiply}_{\mathsf{en}}(\psi_{\mathbf{A},a}, \psi_{\mathbf{A}',a'})$ then we have*

$$\mathsf{Noise}_{\mathbf{A}^\times,a^\times}(\psi_{\mathbf{A}^\times,a^\times}) \leq m \cdot \mathsf{Noise}_{\mathbf{A},a}(\psi_{\mathbf{A},a}) + a \cdot \mathsf{Noise}_{\mathbf{A}',a'}(\psi_{\mathbf{A}',a'})$$

Proof. Given two valid encodings, we have

$$\begin{aligned}
\psi_{\mathbf{A}^\times,a^\times} &= -\widetilde{\mathbf{A}'}^\mathsf{T} \cdot \psi + a \cdot \psi' \\
&= -\widetilde{\mathbf{A}'}^\mathsf{T}\big((\mathbf{A} + a \cdot \mathbf{G})^\mathsf{T}\mathbf{s} + \mathbf{e}\big) + a \cdot \big((\mathbf{A}' + a' \cdot \mathbf{G})^\mathsf{T}\mathbf{s} + \mathbf{e}'\big) \\
&= \left((-\mathbf{A}\widetilde{\mathbf{A}'} - a \cdot \mathbf{A}')^\mathsf{T}\mathbf{s} - \widetilde{\mathbf{A}'}^\mathsf{T}\mathbf{e}\right) + \left((a \cdot \mathbf{A}' + aa' \cdot \mathbf{G})^\mathsf{T}\mathbf{s} + a \cdot \mathbf{e}'\right) \\
&= \big((\underbrace{-\mathbf{A}\widetilde{\mathbf{A}}_2}_{\mathbf{A}^\times}) + \underbrace{aa'}_{a^\times} \cdot \mathbf{G}\big)^\mathsf{T}\mathbf{s} + \big(\underbrace{-\widetilde{\mathbf{A}'}^\mathsf{T}\mathbf{e} + a \cdot \mathbf{e}'}_{\mathbf{e}^\times}\big)
\end{aligned}$$

Thus, from the definition of the noise function, we must bound the noise $\mathbf{e}^\times$. Hence,

$$\left\|\mathbf{e}^\times\right\|_\infty \leq \left\|\widetilde{\mathbf{A}'}^\mathsf{T}\mathbf{e}\right\|_\infty + a \cdot \|\mathbf{e}'\|_\infty \leq m \cdot \|\mathbf{e}\|_\infty + a \cdot \|\mathbf{e}'\|_\infty$$

where the last inequality holds since $\widetilde{\mathbf{A}'} \in \{0,1\}^{m \times m}$. □

Note: This type of homomorphism is different from a standard fully homomorphic encryption (FHE) mainly for the following two reasons.

- To perform multiplicative homomorphism, here we need one of the input values *in clear* but the FHE homomorphic operations are performed without the knowledge of the input values.
- The other big difference is that, here we require the output public key matrices $\mathbf{A}^{+}, \mathbf{A}^{\times}$ to be independent of the input values a_1, a_2. More generally, when given an arbitrary circuit with AND and OR gates along with the matrices corresponding to its input wires, one should be able to determine the matrix corresponding to the output wire without the knowledge of the values of the input wires. But, this property is not present in any of the existing FHE schemes.

2.4.2 Our Public Key Evaluation Algorithm

We define a (public) key evaluation algorithm $\mathsf{Eval}_{\mathsf{pk}}$. The algorithm takes as input a description of the branching program BP, a collection of public keys $\{\mathbf{A}_i\}_{i\in[\ell]}$ (one for each attribute bit $\mathbf{x}_i$), a collection of public keys $\mathbf{V}_{0,i}$ for initial state vector and an auxiliary matrix $\mathbf{A}^{\mathsf{c}}$. The algorithm outputs an "evaluated" public key corresponding to the branching program:

$$\mathsf{Eval}_{\mathsf{pk}}(\mathsf{BP}, \{\mathbf{A}_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}\}_{i\in[5]}, \mathbf{A}^{\mathsf{c}}) \to \mathbf{V}_{\mathsf{BP}}$$

The auxiliary matrix $\mathbf{A}^{\mathsf{c}}$ can be thought of as the public key we use to encode a constant 1. We also define $\mathbf{A}'_i := \mathbf{A}^{\mathsf{c}} - \mathbf{A}_i$, as a public key that will encode $1 - \mathbf{x}_i$. The output $\mathbf{V}_{\mathsf{BP}} \in \mathbb{Z}_q^{n\times m}$ is the homomorphically defined public key $\mathbf{V}_{L,1}$ at position 1 of the state vector at the Lth step of the branching program evaluation.

The algorithm proceeds as follows. Recall the description of the branching program BP represented by tuples $\big(\mathsf{var}(t), \{\gamma_{t,i,0}, \gamma_{t,i,1}\}_{i\in[5]}\big)$ for $t \in [L]$. The initial state vector is always taken to be $\mathbf{v}_0 := [1,0,0,0,0]$. And for $t \in [L]$,

$$\mathbf{v}_t[i] = \mathbf{v}_{t-1}\left[\gamma_{t,i,0}\right] \cdot (1 - x_{\mathsf{var}(t)}) + \mathbf{v}_{t-1}\left[\gamma_{t,i,1}\right] \cdot x_{\mathsf{var}(t)}$$

Our algorithm calculates $\mathbf{V}_{\mathsf{BP}}$ inductively as follows. Assume at time $t-1 \in [L]$, the state public keys $\{\mathbf{V}_{t-1,i}\}_{i\in[5]}$ are already assigned. We assign state public keys $\{\mathbf{V}_{t,i}\}_{i\in[5]}$ at time t as follows.

1. Let $\gamma_0 := \gamma_{t,i,0}$ and $\gamma_1 := \gamma_{t,i,1}$.

2. Let $\mathbf{V}_{t,i} = -\mathbf{A}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} - \mathbf{A}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1}$.

It is important to note that the public key defined at each step of the state vector is *independent of any input attribute vector*. Now, let $\mathbf{V}_{L,1}$ be the public key assigned at position 1 at step L of the branching program. We simply output $\mathbf{V}_{\mathsf{BP}} := \mathbf{V}_{L,1}$.

2.4.3 Our Encoding Evaluation Algorithm

We also define an encoding evaluation algorithm $\mathsf{Eval}_{\mathsf{en}}$ which we will use in the decryption algorithm of our ABE scheme. The algorithm takes as input the description of a branching program BP, an attribute vector $\mathbf{x}$, a set of encodings for the attribute (with corresponding public keys) $\{\mathbf{A}_i, \psi_i := \psi_{\mathbf{A}_i, x_i}\}_{i\in[\ell]}$, encodings of the initial state vector $\{\mathbf{V}_{0,i}, \psi_{0,i} := \psi_{\mathbf{V}_{0,i},\mathbf{v}_0[i]}\}_{i\in[5]}$ and an encoding of a constant "1" $\psi^{\mathsf{c}} := \psi_{\mathbf{A}^{\mathsf{c}},1}$. (From now on, we will use the simplified notations $\psi_i, \psi_{0,i}, \psi^{\mathsf{c}}$ for the encodings). $\mathsf{Eval}_{\mathsf{en}}$ outputs an encoding of the result $y := \mathsf{BP}(\mathbf{x})$ with respect to a homomorphically derived public key $\mathbf{V}_{\mathsf{BP}} := \mathbf{V}_{L,1}$.

$$\mathsf{Eval}_{\mathsf{en}}\big(\mathsf{BP}, \mathbf{x}, \{\mathbf{A}_i, \psi_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}, \psi_{0,i}\}_{i\in[5]}, \mathbf{A}^{\mathsf{c}}, \psi^{\mathsf{c}}\big) \to \psi_{\mathsf{BP}}$$

Recall that for $t \in [L]$, we have for all $i \in [5]$:

$$\mathbf{v}_t[i] = \mathbf{v}_{t-1}\left[\gamma_{t,i,0}\right] \cdot (1 - x_{\mathsf{var}(t)}) + \mathbf{v}_{t-1}\left[\gamma_{t,i,1}\right] \cdot x_{\mathsf{var}(t)}$$

The evaluation algorithm proceeds inductively to update the encoding of the state vector for each step of the branching program. The key idea to obtain the desired noise growth is that we only multiply the *fresh encodings* of the attribute bits with the binary decomposition of the public keys. The result is then added to update the encoding of the state vector. Hence, at each step of the computation the noise in the encodings of the state will only grow by some fixed additive factor.

The algorithm proceeds as follows. We define $\psi'_i := \psi_{\mathbf{A}'_i,(1-x_i)} = (\mathbf{A}'_i + (1-x_i)\cdot\mathbf{G})^{\mathsf{T}}\mathbf{s} + \mathbf{e}'_i$ to denote the encoding of $1-x_i$ with respect to $\mathbf{A}'_i = \mathbf{A}^{\mathsf{c}} - \mathbf{A}_i$. Note that it can be computed using $\mathsf{Add}_{\mathsf{en}}(\psi_{\mathbf{A}^{\mathsf{c}},1}, -\psi_{\mathbf{A}_i,x_i})$. Assume at time $t-1 \in [L]$ we hold encodings of the state vector $\{\psi_{\mathbf{V}_{t-1,i},\mathbf{v}_t[i]}\}_{i\in[5]}$. Now, we compute the encodings of the new state values:

$$\psi_{t,i} = \mathsf{Add}_{\mathsf{en}}\left(\mathsf{Multiply}_{\mathsf{en}}(\psi'_{\mathsf{var}(t)}, \psi_{t-1,\gamma_0}), \mathsf{Multiply}_{\mathsf{en}}(\psi_{\mathsf{var}(t)}, \psi_{t-1,\gamma_1})\right)$$

where $\gamma_0 := \gamma_{t,i,0}$ and $\gamma_1 := \gamma_{t,i,1}$. As we show below (in Lemma 2.4.3), this new encoding has the form $\big(\mathbf{V}_{t,i} + \mathbf{v}_t[i] \cdot \mathbf{G}\big)^{\mathsf{T}}\mathbf{s} + \mathbf{e}_{t,i}$ (for a small enough noise term $\mathbf{e}_{t,i}$).

Finally, let $\psi_{L,1}$ be the encoding obtained at the Lth step corresponding to state value at position "1" by this process. As we show in Lemma 2.4.4, noise term $\mathbf{e}_{\mathsf{BP}}$ has "low" infinity norm enabling correct decryption (Lemma 2.5.1). The algorithm outputs $\psi_{\mathsf{BP}} := \psi_{L,1}$.

Correctness and Analysis

Lemma 2.4.3. *For any valid set of encodings $\psi_{\mathsf{var}(t)}, \psi'_{\mathsf{var}(t)}$ for the bits $x_{\mathsf{var}(t)}, (1 - x_{\mathsf{var}(t)})$ and $\{\psi_{t-1,i}\}_{i\in[5]}$ for the state vector $\mathbf{v}_{t-1}$ at step $t-1$, the output of the function*

$$\mathsf{Add}_{\mathsf{en}}\left(\mathsf{Multiply}_{\mathsf{en}}(\psi'_{\mathsf{var}(t)}, \psi_{t-1,\gamma_0}), \mathsf{Multiply}_{\mathsf{en}}(\psi_{\mathsf{var}(t)}, \psi_{t-1,\gamma_1})\right) \to \psi_{t,i}$$

where $\psi_{t,i} = \big(\mathbf{V}_{t,i} + \mathbf{v}_t[i] \cdot \mathbf{G}\big)^{\mathsf{T}}\mathbf{s} + \mathbf{e}_{t,i}$, for some noise term $\mathbf{e}_{t,i}$.

Proof. Given valid encodings $\psi_{\mathsf{var}(t)}, \psi'_{\mathsf{var}(t)}$ and $\{\psi_{t-1,i}\}_{i\in[5]}$, we have:

$$\begin{aligned}
\psi_{t,i} &= \mathsf{Add}_{\mathsf{en}}\left(\mathsf{Multiply}_{\mathsf{en}}(\psi'_{\mathsf{var}(t)}, \psi_{t-1,\gamma_0}), \mathsf{Multiply}_{\mathsf{en}}(\psi_{\mathsf{var}(t)}, \psi_{t-1,\gamma_1})\right)\\
&= \mathsf{Add}_{\mathsf{en}}\Bigg(\Big[(-\mathbf{A}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} + (\mathbf{v}_t[\gamma_0] \cdot (1 - x_{\mathsf{var}(t)})) \cdot \mathbf{G})^{\mathsf{T}}\mathbf{s} + \mathbf{e}_1)\Big],\\
&\qquad\qquad \Big[(-\mathbf{A}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1} + (\mathbf{v}_t[\gamma_1] \cdot x_{\mathsf{var}(t)}) \cdot \mathbf{G})^{\mathsf{T}}\mathbf{s} + \mathbf{e}_2)\Big]\Bigg)\\
&= \Big[\underbrace{\Big(-\mathbf{A}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} - \mathbf{A}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1}\Big)}_{\mathbf{V}_{t,i}} + \underbrace{\big(\mathbf{v}_t[\gamma_0] \cdot (1 - x_{\mathsf{var}(t)}) + \mathbf{v}_t[\gamma_1] \cdot x_{\mathsf{var}(t)}\big)}_{\mathbf{v}_t[i]} \cdot \mathbf{G}\Big]^{\mathsf{T}}\mathbf{s} + \mathbf{e}_{t,i}
\end{aligned}$$

where the first step follows from the correctness of $\mathsf{Multiply}_{\mathsf{en}}$ algorithm and last step from that of $\mathsf{Add}_{\mathsf{en}}$ with $\mathbf{e}_{t,i} = \mathbf{e}_1 + \mathbf{e}_2$ where $\mathbf{e}_1 = -\left(\widetilde{\mathbf{V}}_{t-1,\gamma_0}\right)^{\mathsf{T}}\mathbf{e}'_{\mathsf{var}(t)} - (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{e}_{t-1,\gamma_0}$ and $\mathbf{e}_2 = -\left(\widetilde{\mathbf{V}}_{t-1,\gamma_1}\right)^{\mathsf{T}}\mathbf{e}_{\mathsf{var}(t)} - x_{\mathsf{var}(t)} \cdot \mathbf{e}_{t-1,\gamma_1}$. □

Lemma 2.4.4. *Let $\mathsf{Eval}_{\mathsf{en}}\big(\mathsf{BP}, \mathbf{x}, \{\mathbf{A}_i, \psi_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}, \psi_{0,i}\}_{i\in[5]}, \mathbf{A}^{\mathsf{c}}, \psi^{\mathsf{c}}\big) \to \psi_{\mathsf{BP}}$ such that all the noise terms, $\big\{\mathsf{Noise}_{\mathbf{A}_i,x_i}(\psi_i)\big\}_{i\in[\ell]}, \mathsf{Noise}_{\mathbf{A}^{\mathsf{c}},1}(\psi^{\mathsf{c}}), \big\{\mathsf{Noise}_{\mathbf{V}_{0,i},\mathbf{v}_0[i]}(\psi_{0,i})\big\}_{i\in[5]}$ are bounded by B, then*

$$\mathsf{Noise}_{\mathbf{V}_{\mathsf{BP}},y}(\psi_{\mathsf{BP}}) \le 3m \cdot L \cdot B + B$$

Proof. We will prove this lemma by induction. That is, we will prove that at any step t,

$$\mathsf{Noise}_{\mathbf{V}_{t,i},\mathbf{v}_t[i]}(\psi_{t,i}) \leq 3m \cdot t \cdot B + B$$

for $i \in [5]$. For the base case, $t = 0$, we operate on *fresh* encodings for the initial state vector $\mathbf{v}_0$. Hence, we have that, $\mathsf{Noise}_{\mathbf{V}_{0,i},\mathbf{v}_0[i]}(\psi_{0,i}) \leq B$, for all $i \in [5]$. Let $\{\psi_{t-1,i}\}_{i\in[5]}$ be the encodings of the state vector $\mathbf{v}_{t-1}$ at step $t-1$ such that

$$\mathsf{Noise}_{\mathbf{V}_{t-1,i},\mathbf{v}_{t-1}[i]}(\psi_{t-1,i}) \leq 3m \cdot (t-1) \cdot B + B$$

for $i \in [5]$. We know that $\psi_{t,i} = \mathsf{Add}_{\mathsf{en}}\Big(\mathsf{Multiply}_{\mathsf{en}}(\psi'_{\mathsf{var}(t)}, \psi_{t-1,\gamma_0}), \mathsf{Multiply}_{\mathsf{en}}(\psi_{\mathsf{var}(t)}, \psi_{t-1,\gamma_1})\Big)$. Hence, from Lemma 2.4.1 and Lemma 2.4.2, we get:

$$\begin{aligned}
\mathsf{Noise}_{\mathbf{V}_{t,i},\mathbf{v}_t[i]}(\psi_{t,i}) &\leq \Big(m \cdot \mathsf{Noise}_{\mathbf{A}'_{\mathsf{var}(t)},(1-x_{\mathsf{var}(t)})}(\psi'_{\mathsf{var}(t)}) + (1 - x_{\mathsf{var}(t)}) \cdot \mathsf{Noise}_{\mathbf{V}_{t-1,\gamma_0},\mathbf{v}_{t-1}[\gamma_0]}\Big) \\
&\qquad + \Big(m \cdot \mathsf{Noise}_{\mathbf{A}_{\mathsf{var}(t)},x_{\mathsf{var}(t)}}(\psi_{\mathsf{var}(t)}) + x_{\mathsf{var}(t)} \cdot \mathsf{Noise}_{\mathbf{V}_{t-1,\gamma_1},\mathbf{v}_{t-1}[\gamma_1]}\Big) \\
&= \big(m \cdot 2B + (1 - x_{\mathsf{var}(t)}) \cdot (3m(t-1)B + B)\big) \\
&\qquad + \big(m \cdot B + x_{\mathsf{var}(t)} \cdot (3m(t-1)B + B)\big) \\
&= 3m \cdot t \cdot B + B
\end{aligned}$$

where

$$\mathsf{Noise}_{\mathbf{A}'_{\mathsf{var}(t)},(1-x_{\mathsf{var}(t)})}(\psi'_{\mathsf{var}(t)}) \leq \mathsf{Noise}_{\mathbf{A}^{\mathsf{c}},1}(\psi^{\mathsf{c}}) + \mathsf{Noise}_{-\mathbf{A}_{\mathsf{var}(t)},-x_{\mathsf{var}(t)}}(-\psi_{\mathsf{var}(t)}) \leq B + B = 2B$$

by Lemma 2.4.1. With ψ_{BP} being an encoding at step L, we have $\mathsf{Noise}_{\mathbf{V}_{\mathsf{BP}},y}(\psi_{\mathsf{BP}}) \leq 3m \cdot L \cdot B + B$. Thus, $\mathsf{Noise}_{\mathbf{V}_{\mathsf{BP}},y}(\psi_{\mathsf{BP}}) = O(m \cdot L \cdot B)$. ☐

2.4.4 Our Simulated Public Key Evaluation Algorithm

Looking ahead, during simulation, we will use a different procedure for assigning public keys to each wire of the input and the state vector. In particular, $\mathbf{A}_i = \mathbf{A} \cdot \mathbf{R}_i - x_i \cdot \mathbf{G}$ for some *shared* public key $\mathbf{A}$ and some low norm matrix $\mathbf{R}_i$. Similarly, the state public keys $\mathbf{V}_{t,i} = \mathbf{A} \cdot \mathbf{R}_{t,i} - \mathbf{v}_t[i] \cdot \mathbf{G}$. The algorithm thus takes as input the description of the branching program BP, the attribute vector $\mathbf{x}$, two collection of low norm matrices $\{\mathbf{R}_i\}, \{\mathbf{R}_{0,i}\}$ corresponding to the input public keys and initial state vector, a low norm matrix $\mathbf{R}^{\mathsf{c}}$ for the public key of constant 1 and a shared matrix $\mathbf{A}$. It outputs a homomorphically derived low norm matrix $\mathbf{R}_{\mathsf{BP}}$.

$$\mathsf{Eval}_{\mathsf{SIM}}(\mathsf{BP}, \mathbf{x}, \{\mathbf{R}_i\}_{i\in[\ell]}, \{\mathbf{R}_{0,i}\}_{i\in[5]}, \mathbf{R}^{\mathsf{c}}, \mathbf{A}) \to \mathbf{R}_{\mathsf{BP}}$$

The algorithm will ensure that the output $\mathbf{R}_{\mathsf{BP}}$ satisfies $\mathbf{A} \cdot \mathbf{R}_{\mathsf{BP}} - \mathsf{BP}(\mathbf{x}) \cdot \mathbf{G} = \mathbf{V}_{\mathsf{BP}}$, where $\mathbf{V}_{\mathsf{BP}}$ is the homomorphically derived public key.

The algorithm proceeds inductively as follows. Assume at time $t-1 \in [L]$, we hold a collection of low norm matrices $\mathbf{R}_{t-1,i}$ and public keys $\mathbf{V}_{t-1,i} = \mathbf{A} \cdot \mathbf{R}_{t-1,i} - \mathbf{v}_t[i] \cdot \mathbf{G}$ for $i \in [5]$ corresponding to the state vector. Let $\mathbf{R}'_i = \mathbf{R}^{\mathsf{c}} - \mathbf{R}_i$ for all $i \in [\ell]$. We show how to derive the low norm matrices $\mathbf{R}_{t,i}$ for all $i \in [5]$:

1. Let $\gamma_0 := \gamma_{t,i,0}$ and $\gamma_1 := \gamma_{t,i,1}$.
2. Compute

$$\mathbf{R}_{t,i} = \Big(-\mathbf{R}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} + (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{R}_{t-1,\gamma_0}\Big) + \Big(-\mathbf{R}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1} + x_{\mathsf{var}(t)} \cdot \mathbf{R}_{t-1,\gamma_1})\Big)$$

Finally, let $\mathbf{R}_{L,1}$ be the matrix obtained at the Lth step corresponding to state value "1" by the above algorithm. Output $\mathbf{R}_{\mathsf{BP}} := \mathbf{R}_{L,1}$. Below, we show that the norm of $\mathbf{R}_{\mathsf{BP}}$ remains small and that homomorphically computed public key $\mathbf{V}_{\mathsf{BP}}$ using $\mathsf{Eval}_{\mathsf{pk}}$ satisfies that $\mathbf{V}_{\mathsf{BP}} = \mathbf{A} \cdot \mathbf{R}_{\mathsf{BP}} - \mathsf{BP}(\mathbf{x}) \cdot \mathbf{G}$.

Lemma 2.4.5 (Correctness of $\mathsf{Eval}_{\mathsf{SIM}}$)**.** *For any set of valid inputs to* $\mathsf{Eval}_{\mathsf{SIM}}$*, we have*

$$\mathsf{Eval}_{\mathsf{SIM}}(\mathsf{BP}, \mathbf{x}, \{\mathbf{R}_i\}_{i\in[\ell]}, \{\mathbf{R}_{0,i}\}_{i\in[5]}, \mathbf{R}^{\mathsf{c}}, \mathbf{A}) \to \mathbf{R}_{\mathsf{BP}}$$

where $\mathbf{V}_{\mathsf{BP}} = \mathbf{A}\mathbf{R}_{\mathsf{BP}} - \mathsf{BP}(\mathbf{x}) \cdot \mathbf{G}$.

Proof. We will prove this lemma by induction. That is, we will prove that at any step t,

$$\mathbf{V}_{t,i} = \mathbf{A}\mathbf{R}_{t,i} - \mathbf{v}_t[i] \cdot \mathbf{G}$$

for any $i \in [5]$. For the base case $t = 0$, since the inputs are valid, we have that $\mathbf{V}_{0,i} = \mathbf{A}\mathbf{R}_{0,i} - \mathbf{v}_0[i] \cdot \mathbf{G}$, for all $i \in [5]$. Let $\mathbf{V}_{t-1,i} = \mathbf{A}\mathbf{R}_{t-1,i} - \mathbf{v}_{t-1}[i] \cdot \mathbf{G}$ for $i \in [5]$. Hence, we get:

$$\begin{aligned}
\mathbf{A}\mathbf{R}_{t,i} &= \Big(-\mathbf{A}\mathbf{R}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} + (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{A}\mathbf{R}_{t-1,\gamma_0}\Big) + \Big(-\mathbf{A}\mathbf{R}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1} + x_{\mathsf{var}(t)} \cdot \mathbf{A}\mathbf{R}_{t-1,\gamma_1})\Big) \\
&= \Big(- \big(\mathbf{A}'_{\mathsf{var}(t)} + (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{G}\big)\widetilde{\mathbf{V}}_{t-1,\gamma_0} + (1 - x_{\mathsf{var}(t)}) \cdot \big(\mathbf{V}_{t-1,\gamma_0} + \mathbf{v}_{t-1}[\gamma_0] \cdot \mathbf{G}\big)\Big) \\
&\quad + \Big(- \big(\mathbf{A}_{\mathsf{var}(t)} + x_{\mathsf{var}(t)} \cdot \mathbf{G}\big)\widetilde{\mathbf{V}}_{t-1,\gamma_1} + x_{\mathsf{var}(t)} \cdot \big(\mathbf{V}_{t-1,\gamma_1} + \mathbf{v}_{t-1}[\gamma_1] \cdot \mathbf{G}\big)\Big) \\
&= \Big(- \mathbf{A}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} - (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{V}_{t-1,\gamma_0} + (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{V}_{t-1,\gamma_0} + \big((1 - x_{\mathsf{var}(t)})\mathbf{v}_{t-1}[\gamma_0]\big) \cdot \mathbf{G}\Big) \\
&\quad + \Big(- \mathbf{A}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1} - x_{\mathsf{var}(t)} \cdot \mathbf{V}_{t-1,\gamma_1} + x_{\mathsf{var}(t)} \cdot \mathbf{V}_{t-1,\gamma_1} + \big(x_{\mathsf{var}(t)}\mathbf{v}_{t-1}[\gamma_1]\big) \cdot \mathbf{G}\Big) \\
&= \big(\underbrace{-\mathbf{A}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} - \mathbf{A}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1}}_{\mathbf{V}_{t,i}}\big) + \big(\underbrace{(1 - x_{\mathsf{var}(t)})\mathbf{v}_{t-1}[\gamma_0] + \big(x_{\mathsf{var}(t)}\mathbf{v}_{t-1}[\gamma_1]}_{\mathbf{v}_t[i]}\big) \cdot \mathbf{G}
\end{aligned}$$

Hence, we have $\mathbf{V}_{t,i} = \mathbf{A}\mathbf{R}_{t,i} - \mathbf{v}_t[i] \cdot \mathbf{G}$. Thus, at the Lth step, we have by induction that

$$\mathbf{V}_{\mathsf{BP}} = \mathbf{V}_{L,1} = \mathbf{A}\mathbf{R}_{L,1-\mathbf{v}_t[i]\cdot\mathbf{G}} = \mathbf{A}\mathbf{R}_{\mathsf{BP}} - \mathbf{v}_t[i] \cdot \mathbf{G}$$

□

Lemma 2.4.6. *Let* $\mathsf{Eval}_{\mathsf{SIM}}(\mathsf{BP}, \mathbf{x}, \{\mathbf{R}_i\}_{i\in[\ell]}, \{\mathbf{R}_{0,i}\}_{i\in[5]}, \mathbf{R}^{\mathsf{c}}, \mathbf{A}) \rightarrow \mathbf{R}_{\mathsf{BP}}$ *such that all the "$\mathbf{R}$" matrices are sampled from* $\{-1,1\}^{m\times m}$, *then*

$$\|\mathbf{R}_{\mathsf{BP}}\|_\infty \leq 3m \cdot L + 1$$

Proof. This proof is very similar to that of Lemma 2.4.4. We will prove this lemma also by induction. That is, we will prove that at any step t,

$$\|\mathbf{R}_{t,i}\|_\infty \leq 3m \cdot t + 1$$

for $i \in [5]$. For the base case, $t = 0$, the input $\mathbf{R}_{0,i}$s are such that, $\|\mathbf{R}_{t,0}\|_\infty = 1$, for all $i \in [5]$. Let $\|\mathbf{R}_{t-1,i}\|_\infty \leq 3m \cdot (t-1) + 1$ for $i \in [5]$. We know that

$$\mathbf{R}_{t,i} = \left(-\mathbf{R}'_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_0} + (1 - x_{\mathsf{var}(t)}) \cdot \mathbf{R}_{t-1,\gamma_0}\right) + \left(-\mathbf{R}_{\mathsf{var}(t)}\widetilde{\mathbf{V}}_{t-1,\gamma_1} + x_{\mathsf{var}(t)} \cdot \mathbf{R}_{t-1,\gamma_1})\right)$$

Hence, we have:

$$\begin{aligned}
\|\mathbf{R}_{t,i}\|_\infty &\leq \left(m \cdot \left\|\widetilde{\mathbf{V}}_{t-1,\gamma_0}\right\|_\infty \cdot \left\|\mathbf{R}'_{\mathsf{var}(t)}\right\|_\infty + (1 - x_{\mathsf{var}(t)}) \cdot \|\mathbf{R}_{t-1,\gamma_0}\|_\infty\right) \\
&\quad + \left(m \cdot \left\|\widetilde{\mathbf{V}}_{t-1,\gamma_0}\right\|_\infty \cdot \left\|\mathbf{R}_{\mathsf{var}(t)}\right\|_\infty + x_{\mathsf{var}(t)} \cdot \|\mathbf{R}_{t-1,\gamma_1}\|_\infty\right) \\
&= \left(m \cdot 1 \cdot 2 + (1 - x_{\mathsf{var}(t)}) \cdot 3m \cdot (t-1)\right) + \left(m \cdot 1 \cdot 1 + x_{\mathsf{var}(t)} \cdot 3m \cdot (t-1)\right) \\
&= 3m \cdot t + 1
\end{aligned}$$

where $\|\mathbf{R}'_i\|_\infty \leq \|\mathbf{R}^{\mathsf{c}} + \mathbf{R}_i\|_\infty \leq \|\mathbf{R}^{\mathsf{c}}\|_\infty + \|\mathbf{R}_i\|_\infty \leq 1 + 1 = 2$. With $\mathbf{R}_{\mathsf{BP}}$ being at step L, we have $\|\mathbf{R}_{\mathsf{BP}}\|_\infty \leq 3m \cdot L + 1$. Thus, $\|\mathbf{R}_{\mathsf{BP}}\|_\infty = O(m \cdot L)$. □

2.5 Our Attribute-Based Encryption

In this section we describe our attribute-based encryption scheme for branching programs. We present the scheme for a bounded length branching programs, but note that we can trivially support unbounded length by setting modulo q to a small superpolynomial. For a family of branching programs of length bounded by L and input space $\{0,1\}^\ell$, we define the $\mathcal{ABE}$ algorithms ($\mathsf{Params}, \mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}$) as follows.

- $\mathsf{Params}(1^\lambda, 1^L)$: For a security parameter λ and length bound L, let the LWE dimension be $n = n(\lambda)$ and let the LWE modulus be $q = q(n, L)$. Let χ be an error distribution over $\mathbb{Z}$ and let $B = B(n)$ be an error bound. We additionally choose two Gaussian parameters: a "small" Gaussian parameter $s = s(n)$ and a "large" Gaussian parameter $\alpha = \alpha(n)$. Both these parameters are polynomially bounded (in λ, L). The public parameters $\mathsf{pp} = (\lambda, L, n, q, m, \chi, B, s, \alpha)$ are implicitly given as input to all the algorithms below.

- $\mathsf{Setup}(1^\ell)$: The setup algorithm takes as input the length of the attribute vector ℓ.
  1. Sample a matrix with a trapdoor: $(\mathbf{A}, \mathbf{T_A}) \leftarrow \mathsf{TrapSamp}(1^n, 1^m, q)$.
  2. Let $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$ be the primitive matrix with the public trapdoor basis $\mathbf{T_G}$.
  3. Choose $\ell + 6$ matrices $\{\mathbf{A}_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}\}_{i\in[5]}, \mathbf{A}^\mathsf{c}$ at random from $\mathbb{Z}_q^{n\times m}$. First, ℓ matrices form the LWE "public keys" for the bits of attribute vector, next 5 form the "public keys" for the initial configuration of the state vector, and the last matrix as a "public key" for a constant 1.
  4. Choose a vector $\mathbf{u} \in \mathbb{Z}_q^n$ at random.
  5. Output the master public key
  $$\mathsf{mpk} := \big(\mathbf{A}, \mathbf{A}^\mathsf{c}, \{\mathbf{A}_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}\}_{i\in[5]}, \mathbf{G}, \mathbf{u}\big)$$
  and the master secret key $\mathsf{msk} := (\mathbf{T_A}, \mathsf{mpk})$.

- $\mathsf{Enc}(\mathsf{mpk}, \mathbf{x}, \mu)$: The encryption algorithm takes as input the master public key mpk, the attribute vector $\mathbf{x} \in \{0,1\}^\ell$ and a message μ.
  1. Choose an LWE secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ at random.
  2. Choose noise vector $\mathbf{e} \xleftarrow{\$} \chi^m$ and compute $\psi_0 = \mathbf{A}^\mathsf{T}\mathbf{s} + \mathbf{e}$.
  3. Choose a random matrix $\mathbf{R}^\mathsf{c} \leftarrow \{-1,1\}^{m\times m}$ and let $\mathbf{e}^\mathsf{c} = (\mathbf{R}^\mathsf{c})^\mathsf{T}\mathbf{e}$. Now, compute an encoding of a constant 1:
  $$\psi^\mathsf{c} = (\mathbf{A}^\mathsf{c} + \mathbf{G})^\mathsf{T}\,\mathbf{s} + \mathbf{e}^\mathsf{c}$$
  4. Encode each bit $i \in [\ell]$ of the attribute vector:
     (a) Choose random matrices $\mathbf{R}_i \leftarrow \{-1,1\}^{m\times m}$ and let $\mathbf{e}_i = \mathbf{R}_i^\mathsf{T}\mathbf{e}$.
     (b) Compute $\psi_i = (\mathbf{A}_i + x_i \cdot \mathbf{G})^\mathsf{T}\mathbf{s} + \mathbf{e}_i$.
  5. Encode the initial state configuration vector $\mathbf{v}_0 = [1,0,0,0,0]$: for all $i \in [5]$,

(a) Choose a random matrix $\mathbf{R}_{0,i} \leftarrow \{-1,1\}^{m\times m}$ and let $\mathbf{e}_{0,i} = \mathbf{R}_{0,i}^{\mathsf{T}}\mathbf{e}$.

(b) Compute $\psi_{0,i} = (\mathbf{V}_{0,i} + \mathbf{v}_0[i]\cdot\mathbf{G})^{\mathsf{T}}\mathbf{s} + \mathbf{e}_{0,i}$.

6. Encrypt the message μ as $\tau = \mathbf{u}^{\mathsf{T}}\mathbf{s} + e + \lfloor q/2 \rceil \mu$, where $e \leftarrow \chi$.

7. Output the ciphertext

$$\mathsf{ct}_{\mathbf{x}} = \left(\mathbf{x}, \psi_0, \psi^{\mathsf{c}}, \{\psi_i\}_{i\in[\ell]}, \{\psi_{0,i}\}_{i\in[5]}, \tau\right)$$

- $\mathsf{KeyGen}(\mathsf{msk}, \mathsf{BP})$: The key-generation algorithm takes as input the master secret key msk and a description of a branching program:

$$\mathsf{BP} := \left(\mathbf{v}_0, \left\{\mathsf{var}(t), \{\gamma_{t,i,0}, \gamma_{t,i,1}\}_{i\in[5]}\right\}_{t\in[L]}\right)$$

The secret key $\mathsf{sk}_{\mathsf{BP}}$ is computed as follows.

1. Homomorphically compute a "public key" matrix associated with the branching program:
$$\mathbf{V}_{\mathsf{BP}} \leftarrow \mathsf{Eval}_{\mathsf{pk}}(\mathsf{BP}, \{\mathbf{A}_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}\}_{i\in[5]}, \mathbf{A}^{\mathsf{c}})$$

2. Let $\mathbf{F} = [\mathbf{A}||(\mathbf{V}_{\mathsf{BP}} + \mathbf{G})] \in \mathbb{Z}_q^{n\times 2m}$. Compute $\mathbf{r}_{\mathsf{output}} \leftarrow \mathsf{SampleLeft}(\mathbf{A}, (\mathbf{V}_{\mathsf{BP}} + \mathbf{G}), \mathbf{T}_{\mathbf{A}}, \mathbf{u}, \alpha)$ such that $\mathbf{F}\cdot\mathbf{r}_{\mathsf{output}} = \mathbf{u}$.

3. Output the secret key for the branching program as

$$\mathsf{sk}_{\mathsf{BP}} := (\mathsf{BP}, \mathbf{r}_{\mathsf{output}})$$

- $\mathsf{Dec}(\mathsf{sk}_{\mathsf{BP}}, \mathsf{ct}_{\mathbf{x}})$: The decryption algorithm takes as input the secret key for a branching program $\mathsf{sk}_{\mathsf{BP}}$ and a ciphertext $\mathsf{ct}_{\mathbf{x}}$. If $\mathsf{BP}(\mathbf{x}) = 0$, output $\perp$. Otherwise,

1. Homomorphically compute the encoding of the result $\mathsf{BP}(\mathbf{x})$ associated with the public key of the branching program:

$$\psi_{\mathsf{BP}} \leftarrow \mathsf{Eval}_{\mathsf{en}}(\mathsf{BP}, \mathbf{x}, \{\mathbf{A}_i, \psi_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}, \psi_{0,i}\}_{i\in[5]}, (\mathbf{A}^{\mathsf{c}}, \psi^{\mathsf{c}}))$$

2. Finally, compute $\phi = \mathbf{r}_{\mathsf{output}}^{\mathsf{T}} \cdot [\psi||\psi_{\mathsf{BP}}]$. As we show in Lemma 2.5.1, $\phi = \mathbf{u}^{\mathsf{T}}\mathbf{s} + \lfloor q/2 \rceil \mu + e_\phi \pmod q$, for a short e_ϕ.

3. Output $\mu = 0$ if $|\tau - \phi| < q/4$ and $\mu = 1$ otherwise.

2.5.1 Correctness

Lemma 2.5.1. *Let $\mathcal{BP}$ be a family of width-5 permutation branching programs with their length bounded by L and let $\mathcal{ABE} = (\mathsf{Params}, \mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be our attribute-based encryption scheme. For a LWE dimension $n = n(\lambda)$, the parameters for $\mathcal{ABE}$ are instantiated as follows (according to Section 2.6):*

$$\chi = D_{\mathbb{Z},\sqrt{n}} \qquad B = O(n) \qquad m = O(n \log q)$$
$$q = \tilde{O}(n^7 \cdot L^2) \qquad \alpha = \tilde{O}(n \log q)^2 \cdot L$$

then the scheme $\mathcal{ABE}$ is correct, according to the definition in Section 2.3.2.

Proof. We have to show that the decryption algorithm outputs the correct message μ, given a valid set of a secret key and a ciphertext.

From Lemma 2.4.3, we have that $\psi_{\mathsf{BP}} = (\mathbf{V}_{\mathsf{BP}} + \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}_{\mathsf{BP}}$ since $\mathsf{BP}(\mathbf{x}) = 1$. Also, from Lemma 2.4.4, we know that $\|\mathbf{e}_{\mathsf{BP}}\|_\infty = O(m \cdot L \cdot (m \cdot B)) = O(m^2 \cdot L \cdot B)$ since our input encodings have noise terms bounded by $m \cdot B$. Thus, the noise term in ϕ is bounded by:

$$\begin{aligned} \|e_\phi\|_\infty &= m \cdot \big(\mathsf{Noise}_{\mathbf{A},0}(\psi) + \mathsf{Noise}_{\mathbf{V}_{\mathsf{BP}},1}(\psi_{\mathsf{BP}})\big) \cdot \|\mathbf{r}_{\mathsf{output}}\|_\infty \\ &= m \cdot (B + O(m^2 \cdot L \cdot B)) \cdot \tilde{O}(n \log q)^2 \cdot L\sqrt{m} \\ &= O\big((n \log q)^6 \cdot L^2 \cdot B\big) \end{aligned}$$

where $m = O(n \log q)$ and $\|\mathbf{r}_{\mathsf{output}}\|_\infty \le \alpha\sqrt{m} = \tilde{O}(n \log q)^2 \cdot L\sqrt{m}$ according to Section 2.6. Hence, we have

$$|\tau - \phi| \le \|e\|_\infty + \|e_\phi\|_\infty = O\big((n \log q)^6 \cdot L^2 \cdot B\big) \le q/4$$

Clearly, the last inequality is satisfied when $q = \tilde{O}(n^7 \cdot L^2)$. Hence, the decryption proceeds correctly outputting the correct μ. □

2.5.2 Security Proof

Theorem 2.5.2. *For any ℓ and any length bound L, $\mathcal{ABE}$ scheme defined above satisfies selective security game from Definition 2.3.3 for any family of branching programs BP of length L with ℓ-bit inputs, assuming hardness of $\mathsf{dLWE}_{n,q,\chi}$ for sufficiently large $n = \mathrm{poly}(\lambda), q = \tilde{O}(n^7 \cdot L^2)$ and $\mathrm{poly}(n)$ bounded error distribution χ. Moreover, the size of the secret keys grows polynomially with L (and independent of the width of BP).*

Proof. We define a series of hybrid games, where the first and the last games correspond to the real experiments encrypting messages μ_0, μ_1 respectively. We show that these games are indistinguishable except with negligible probability. Recall that in a selective security game, the challenge attribute vector $\mathbf{x}^*$ is declared before the Setup algorithm and all the secret key queries that adversary makes must satisfy $\mathsf{BP}(\mathbf{x}^*) = 0$. First, we define auxiliary simulated $\mathcal{ABE}^*$ algorithms.

- $\mathsf{Setup}^*(1^\lambda, 1^\ell, 1^L, \mathbf{x}^*)$: The simulated setup algorithm takes as input the security parameter λ, the challenge attribute vector $\mathbf{x}^*$, its length ℓ and the maximum length of the branching program L.
  1. Choose a random matrix $\mathbf{A} \leftarrow \mathbb{Z}_q^{n\times m}$ and a vector $\mathbf{u}$ at random.
  2. Let $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$ be the primitive matrix with the public trapdoor basis $\mathbf{T_G}$.
  3. Choose $\ell + 6$ random matrices $\{\mathbf{R}_i\}_{i\in[\ell]}, \{\mathbf{R}_{0,i}\}_{i\in[5]}, \mathbf{R}^{\mathsf{c}}$ from $\{-1, 1\}^{m\times m}$ and set
     (a) $\mathbf{A}_i = \mathbf{A} \cdot \mathbf{R}_i - x^*\mathbf{G}$ for $i \in [\ell]$,
     (b) $\mathbf{V}_{0,i} = \mathbf{A} \cdot \mathbf{R}_{0,i} - \mathbf{v}_0[i] \cdot \mathbf{G}$ for $i \in [5]$ where $\mathbf{v}_0 = [1, 0, 0, 0, 0]$,
     (c) $\mathbf{A}^{\mathsf{c}} = \mathbf{A} \cdot \mathbf{R}^{\mathsf{c}} - \mathbf{G}$.
  4. Output the master public key

     $$\mathsf{mpk} := \big(\mathbf{A}, \mathbf{A}^{\mathsf{c}}, \{\mathbf{A}_i\}_{i\in[\ell]}, \{\mathbf{V}_{0,i}\}_{i\in[5]}, \mathbf{G}, \mathbf{u}\big)$$

     and the secret key

     $$\mathsf{msk} := \big(\mathbf{x}^*, \mathbf{A}, \mathbf{R}^{\mathsf{c}}, \{\mathbf{R}_i\}_{i\in[\ell]}, \{\mathbf{R}_{0,i}\}_{i\in[5]}\big)$$

- $\mathsf{Enc}^*(\mathsf{mpk}, \mathbf{x}^*, \mu)$: The simulated encryption algorithm takes as input $\mathsf{mpk}, \mathbf{x}^*$ and the message μ. It computes the ciphertext using the knowledge of short matrices $\{\mathbf{R}_i\}, \{\mathbf{R}_{0,i}\}, \mathbf{R}^{\mathsf{c}}$ as follows.
  1. Choose a vector $\mathbf{s} \in \mathbb{Z}_q^n$ at random.
  2. Choose noise vector $\mathbf{e} \xleftarrow{\$} \chi^m$ and compute $\psi_0 = \mathbf{A}^{\mathsf{T}}\mathbf{s} + \mathbf{e}$.
  3. Compute an encoding of an identity as $\psi^{\mathsf{c}} = (\mathbf{A}^{\mathsf{c}})^{\mathsf{T}}\,\mathbf{s} + (\mathbf{R}^{\mathsf{c}})^{\mathsf{T}}\mathbf{e}$.
  4. For all bits of the attribute vector $i \in [\ell]$ compute

     $$\psi_i = (\mathbf{A}_i + x_i \cdot \mathbf{G})^{\mathsf{T}}\mathbf{s} + \mathbf{R}_i^{\mathsf{T}}\mathbf{e}$$

5. For all $i \in [5]$, encode the bits of the initial state configuration vector $\mathbf{v}_0 = [1, 0, 0, 0, 0]$
$$\psi_{0,i} = (\mathbf{V}_{0,i} + \mathbf{v}_0[i] \cdot \mathbf{G})^\intercal \mathbf{s} + \mathbf{R}_{0,i}^\intercal \mathbf{e}$$
6. Encrypt the message μ as $\tau = \mathbf{u}^\intercal \mathbf{s} + e + \lfloor q/2 \rceil \mu$, where $e \leftarrow \chi$.
7. Output the ciphertext
$$\mathsf{ct} = \big(\mathbf{x}, \psi_0, \{\psi_i\}_{i \in [\ell]}, \psi^{\mathsf{c}}, \{\psi_{0,i}\}_{i \in [5]}, \tau\big)$$

- $\mathsf{KeyGen}^*(\mathsf{msk}, \mathsf{BP})$: The simulated key-generation algorithm takes as input the master secret key msk and the description of the branching program BP. It computes the secret key $\mathsf{sk}_{\mathsf{BP}}$ as follows.

1. Obtain a short homomorphically derived matrix associated with the output public key of the branching program:
$$\mathbf{R}_{\mathsf{BP}} \leftarrow \mathsf{Eval}_{\mathsf{SIM}}\big(\mathsf{BP}, \mathbf{x}^*, \{\mathbf{R}_i\}_{i \in [\ell]}, \{\mathbf{R}_{0,i}\}_{i \in [5]}, \mathbf{R}^{\mathsf{c}}, \mathbf{A}\big)$$
2. By the correctness of $\mathsf{Eval}_{\mathsf{SIM}}$, we have $\mathbf{V}_{\mathsf{BP}} = \mathbf{A}\mathbf{R}_{\mathsf{BP}} - \mathsf{BP}(\mathbf{x}^*) \cdot \mathbf{G}$. Let $\mathbf{F} = [\mathbf{A} || (\mathbf{V}_{\mathsf{BP}} + \mathbf{G})] \in \mathbb{Z}_q^{n \times 2m}$. Compute $\mathbf{r}_{\mathsf{output}} \leftarrow \mathsf{SampleRight}(\mathbf{A}, \mathbf{G}, \mathbf{R}_{\mathsf{BP}}, \mathbf{T}_{\mathbf{G}}, \mathbf{u}, \alpha)$ such that $\mathbf{F} \cdot \mathbf{r}_{\mathsf{output}} = \mathbf{u}$ (this step relies on the fact that $\mathsf{BP}(\mathbf{x}^*) = 0$).
3. Output the secret key for the branching program
$$\mathsf{sk}_{\mathsf{BP}} := (\mathsf{BP}, \mathbf{r}_{\mathsf{output}})$$

Game Sequence. We now define a series of games and then prove that all games **Game i** and **Game i+1** are either statistically or computationally indistinguishable.

- **Game 0**: The challenger runs the real ABE algorithms and encrypts message μ_0 for the challenge index $\mathbf{x}^*$.
- **Game 1**: The challenger runs the simulated ABE algorithms $\mathsf{Setup}^*, \mathsf{KeyGen}^*, \mathsf{Enc}^*$ and encrypts message μ_0 for the challenge index $\mathbf{x}^*$.
- **Game 2**: The challenger runs the simulated ABE algorithms $\mathsf{Setup}^*, \mathsf{KeyGen}^*$, but chooses a uniformly random element of the ciphertext space for the challenge index $\mathbf{x}^*$.
- **Game 3**: The challenger runs the simulated ABE algorithms $\mathsf{Setup}^*, \mathsf{KeyGen}^*, \mathsf{Enc}^*$ and encrypts message μ_1 for the challenge index $\mathbf{x}^*$.

- **Game 4**: The challenger runs the real ABE algorithms and encrypts message μ_1 for the challenge index $\mathbf{x}^*$.

Lemma 2.5.3. *The view of an adversary in* **Game 0** *is statistically indistinguishable from* **Game 1**. *Similarly, the view of an adversary in* **Game 4** *is statistically indistinguishable from* **Game 3**.

Proof. We prove for the case of **Game 0** and **Game 1**, as the other case is identical. First, note the differences between the games:

- In **Game 0**, matrix $\mathbf{A}$ is sampled using $\mathsf{TrapSamp}$ algorithm and matrices $\mathbf{A}_i, \mathbf{A}^{\mathsf{c}}, \mathbf{V}_{0,j} \in \mathbb{Z}_q^{n\times m}$ are randomly chosen for $i \in [\ell], j \in [5]$. In **Game 1**, matrix $\mathbf{A} \in \mathbb{Z}_p^{n\times m}$ is chosen uniformly at random and matrices $\mathbf{A}_i = \mathbf{A}\mathbf{R}_i - x_i^* \cdot \mathbf{G}$, $\mathbf{A}^{\mathsf{c}} = \mathbf{A}\mathbf{R}^{\mathsf{c}} - \mathbf{G}$, $\mathbf{V}_{0,j} = \mathbf{A}\mathbf{R}_{0,j} - \mathbf{v}_0[j] \cdot \mathbf{G}$ for randomly chosen $\mathbf{R}_i, \mathbf{R}^{\mathsf{c}}, \mathbf{R}_{0,j} \in \{-1, 1\}^{m\times m}$.

- In **Game 0**, each ciphertext component is computed as:

$$\begin{aligned} \psi_i &= (\mathbf{A}_i + x_i^* \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}_i = (\mathbf{A}_i + x_i^* \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{R}_i^{\intercal}\mathbf{e} \\ \psi^{\mathsf{c}} &= (\mathbf{A}^{\mathsf{c}} + \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}^1 = (\mathbf{A}^{\mathsf{c}} + \mathbf{G})^{\intercal}\mathbf{s} + (\mathbf{R}^{\mathsf{c}})^{\intercal}\mathbf{e} \\ \psi_{0,j} &= (\mathbf{V}_{0,j} + \mathbf{v}_0[j] \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{e}_i = (\mathbf{V}_{0,j} + \mathbf{v}_0[j] \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{R}_{0,j}^{\intercal}\mathbf{e} \end{aligned}$$

  On the other hand, in **Game 1** each ciphertext component is computed as:

$$\psi_i = (\mathbf{A}_i + x_i^* \cdot \mathbf{G})^{\intercal}\mathbf{s} + \mathbf{R}_i^{\intercal}\mathbf{e} = (\mathbf{A}\mathbf{R}_i)^{\intercal}\mathbf{s} + \mathbf{R}_i^{\intercal}\mathbf{e} = \mathbf{R}_i^{\intercal}\big(\mathbf{A}^{\intercal}\mathbf{s} + \mathbf{e}\big)$$

  Similarly, $\psi^{\mathsf{c}} = (\mathbf{R}^{\mathsf{c}})^{\intercal}(\mathbf{A}^{\intercal}\mathbf{s} + \mathbf{e})$ and $\psi_{0,j} = \mathbf{R}_{0,j}^{\intercal}(\mathbf{A}\mathbf{s} + \mathbf{e})$.

- Finally, in **Game 0** the vector $\mathbf{r}_{\mathsf{output}}$ is sampled using $\mathsf{SampleLeft}$, whereas in **Game 1** it is sampled using $\mathsf{SampleRight}$ algorithm.

For sufficiently large α (See Section-2.6), the distributions produced in two games are statistically indistinguishable. This follows readily from [6, Lemma 4.3], Theorem-2.3.2 and Theorem-2.3.3. We will provide the proof here for completeness.

We would like to prove that the tuple $\mathbf{A}, \{\mathbf{A}_i, \psi_i\}_{i\in[\ell]}, \mathbf{A}^{\mathsf{c}}, \psi^{\mathsf{c}}, \{\mathbf{V}_{0,j}, \psi_{0,j}\}_{j\in[5]}$ in **Game 0** is statistically indistinguishable from the set from **Game 1**. The generalisation of left-over hash lemma [80, 5] states that, for two matrices $\mathbf{R}_i \xleftarrow{\$} \{-1, 1\}^{m\times m}, \mathbf{A}, \mathbf{A}_i \xleftarrow{\$} \mathbb{Z}_q^{n\times m}$ and any vector $\mathbf{e} \in \mathbb{Z}_q^m$, the following is true, when q is square-free (q does not have a square of a

prime number as its factor): $(\mathbf{A}, \mathbf{A}\mathbf{R}_i, \mathbf{R}_i^\intercal \mathbf{e},) \approx_s (\mathbf{A}, \mathbf{A}_i, \mathbf{R}_i^\intercal \mathbf{e})$. With the matrices $\mathbf{R}_i, \mathbf{R}_{0,j}$ independently chosen from $\{-1, 1\}^{m\times m}$ we can extend this to have:

$$\begin{aligned}&\left(\mathbf{A}, \left(\mathbf{A}\mathbf{R}_i, \mathbf{R}_i^\intercal \mathbf{e}\right), \left(\mathbf{A}\mathbf{R}^\mathsf{c}, (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right), \left(\mathbf{A}\mathbf{R}_{0,j}, \mathbf{R}_{0,j}^\intercal \mathbf{e}\right)\right)\\ &\quad\approx_s \left(\mathbf{A}, \left(\mathbf{A}_i, \mathbf{R}_i^\intercal \mathbf{e}\right), \left(\mathbf{A}^\mathsf{c}, (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right), \left(\mathbf{V}_{0,j}, \mathbf{R}_{0,j}^\intercal \mathbf{e}\right)\right)\end{aligned}$$

Hence, for every fixed matrix $\mathbf{G} \in \mathbb{Z}_q^{n\times m}$ and every bit $x_i^*, \mathbf{v}_0[j] \in \{0, 1\}$,

$$\begin{aligned}&\left(\mathbf{A}, \left(\mathbf{A}\mathbf{R}_i - x_i^* \cdot \mathbf{G}, \mathbf{R}_i^\intercal \mathbf{e}\right), \left(\mathbf{A}\mathbf{R}^\mathsf{c} - \mathbf{G}, (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right), \left(\mathbf{A}\mathbf{R}_{0,j} - \mathbf{v}_0[j], \mathbf{R}_{0,j}^\intercal \mathbf{e}\right)\right)\\ &\qquad\approx_s \left(\mathbf{A}, \left(\mathbf{A}_i, \mathbf{R}_i^\intercal \mathbf{e}\right), \left(\mathbf{A}^\mathsf{c}, (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right), \left(\mathbf{V}_{0,j}, \mathbf{R}_{0,j}^\intercal \mathbf{e}\right)\right)\end{aligned}$$

Now, we can extend this statistical indistinguishability to the joint distribution of these tuples for all $i \in [\ell], j \in [5]$, since the matrices $\mathbf{R}_i, \mathbf{R}_{0,j}$ are independently chosen from $\{-1, 1\}^{m\times m}, \forall i \in [\ell], j \in [5]$. Thus,

$$\begin{aligned}&\left(\mathbf{A}, \left(\{\mathbf{A}\mathbf{R}_i - x_i^* \cdot \mathbf{G}, \mathbf{R}_i^\intercal \mathbf{e}\}_{i\in[\ell]}\right), \left(\mathbf{A}\mathbf{R}^\mathsf{c} - \mathbf{G}, (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right), \left(\{\mathbf{A}\mathbf{R}_{0,j} - \mathbf{v}_0[j], \mathbf{R}_{0,j}^\intercal \mathbf{e}\}_{j\in[5]}\right)\right)\\ &\qquad\approx_s \left(\mathbf{A}, \left(\{\mathbf{A}_i, \mathbf{R}_i^\intercal \mathbf{e}\}_{l\in[\ell]}\right), \left(\mathbf{A}^\mathsf{c}, (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right), \left(\{\mathbf{V}_{0,j}, \mathbf{R}_{0,j}^\intercal \mathbf{e}\}_{j\in[5]}\right)\right)\end{aligned}$$

Also, due to the fact that applying any function to two statistically indistinguishable entities results in entities which are atleast as statistically indistinguishable as the original pair, we eventually get:

$$\begin{aligned}&\Big(\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e}, \left(\{\mathbf{A}\mathbf{R}_i - x_i^* \cdot \mathbf{G}, (\mathbf{A}\mathbf{R}_i)^\intercal \mathbf{s} + \mathbf{R}_i^\intercal \mathbf{e}\}_{i\in[\ell]}\right), \left(\mathbf{A}\mathbf{R}^\mathsf{c} - \mathbf{G}, (\mathbf{A}\mathbf{R}^\mathsf{c})^\intercal + (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\right),\\ &\qquad\qquad \left(\{\mathbf{A}\mathbf{R}_{0,j} - \mathbf{v}_0[j], (\mathbf{A}\mathbf{R}_{0,j})^\intercal + \mathbf{R}_{0,j}^\intercal \mathbf{e}\}_{j\in[5]}\right)\Big)\\ &\approx_s \Big(\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e}, \left(\mathbf{A}_i, (\mathbf{A}_i + x_i \cdot \mathbf{G})^\intercal \mathbf{s} + \mathbf{R}_i^\intercal \mathbf{e}\right), \left(\{\mathbf{A}^\mathsf{c}, (\mathbf{A}^\mathsf{c} + \mathbf{G})^\intercal \mathbf{s} + (\mathbf{R}^\mathsf{c})^\intercal \mathbf{e}\}_{i\in[\ell]}\right),\\ &\qquad\qquad \left(\{\mathbf{V}_{0,j}, (\mathbf{V}_{0,j} + \mathbf{v}_0[j] \cdot \mathbf{G})^\intercal \mathbf{s} + \mathbf{R}_{0,j}^\intercal \mathbf{e}\}_{j\in[5]}\right)\Big)\end{aligned}$$

Thus, we can conclude that the public parameters in **Game 0** are statistically indistinguishable from those in **Game 1**, and that the output of Enc is statistically indistinguishable from that of Enc*. When the "large" Gaussian parameter α is chosen appropriately (as discussed in 2.6), the output of the KeyGen and KeyGen* algorithms are also statistically indistinguishable. Thus, the view of an adversary in **Game 0** is statistically indistinguishable from the view in **Game 1**.

□

Lemma 2.5.4. *If the decisional* LWE *assumption holds, then the view of an adversary in* **Game 1** *is computationally indistinguishable from* **Game 2**. *Similarly, if the decisional* LWE *assumption holds, then the view of an adversary in* **Game 3** *is computationally indistinguishable from* **Game 2**.

Proof. Assume there exist an adversary Adv that distinguishes between **Game 1** and **Game 2**. We show how to break LWE problem given a challenge $\{(\mathbf{a}_i, y_i)\}_{i\in[m+1]}$ where each y_i is either a random sample in $\mathbb{Z}_q$ or $\mathbf{a}_i^\intercal \cdot \mathbf{s} + e_i$ (for a fixed, random $\mathbf{s} \in \mathbb{Z}_q^n$ and a noise term sampled from the error distribution $e_i \leftarrow \chi$). Let $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m] \in \mathbb{Z}_q^{n\times m}$ and $\mathbf{u} = \mathbf{a}_{m+1}$. Let $\psi_0^* = [y_1, y_2, \ldots, y_m]$ and $\tau = y_{m+1} + \mu \lfloor q/2 \rfloor$.

Now, run the simulated Setup^* algorithm where $\mathbf{A}, \mathbf{u}$ are as defined above. Run the simulated KeyGen^* algorithm. Finally, to simulate the challenge ciphertext set ψ_0^*, τ as defined above and compute

$$\psi_i = \mathbf{R}_i^\intercal \cdot \psi_0^* = \mathbf{R}_i^\intercal \big(\mathbf{A}^\intercal \mathbf{s} + \mathbf{e}\big)$$

for $i \in [\ell]$. Similarly, $\psi^{\mathsf{c}} = (\mathbf{R}^{\mathsf{c}})^\intercal (\mathbf{A}^\intercal \mathbf{s} + \mathbf{e})$ and $\psi_{0,j} = \mathbf{R}_{0,j}^\intercal (\mathbf{A}^\intercal \mathbf{s} + \mathbf{e})$, for $j \in [5]$. Note that if y_i's are LWE samples, then this corresponds exactly to the **Game 1**. Otherwise, the ciphertext corresponds to an independent random sample as in **Game 2** by the left-over hash lemma. Thus, an adversary which distinguishes between **Game 1** and **Game 2** can also be used to break the decisional LWE assumption with almost the same advantage. The computational indistinguishability of **Game 3** and **Game 2** follows from the same argument. □

Thus, **Game 0** and **Game 4** are computationally indistinguishable by the standard hybrid argument and hence no adversary can distinguish between encryptions of μ_0 and μ_1 with non-negligible advantage establishing the selective security of our ABE scheme. □

2.6 Parameter Selection

This section provides a detailed description on the selection of parameters for our scheme, so that both correctness (see Lemma 2.5.1) and security (see Theorem 2.5.2) of our scheme are satisfied.

For a family of width-5 permutation branching programs $\mathcal{BP}$ of bounded length L, with the LWE dimension n, the parameters can be chosen as follows:

- The error distribution $\chi = D_{\mathbb{Z},\sqrt{n}}$ with parameter $\sigma = \sqrt{n}$. And, the error bound $B = O(\sigma\sqrt{n}) = O(n)$.

From now, we will consider the LWE modulus parameter $q = q(n, L)$, without instantiating it, to calculate the other parameters m, s, α. Later, we will instantiate q with a value which would make m, s, α satisfy the correctness and security properties.

- The parameter $m = O(n \log q)$.
- The "small" Gaussian parameter s is chosen to be $O(\sqrt{n \log q})$.
- Now, let us calculate the value of the "large" Gaussian parameter $\alpha = \alpha(n, L)$. We should choose α such that the output of the $\mathsf{SampleLeft}$ and the $\mathsf{SampleRight}$ algorithms are statistically indistinguishable from each other, when provided with the same set of inputs $\mathbf{F}$ and $\mathbf{u}$.

  The $\mathsf{SampleRight}$ algorithm (Algorithm 2.5) requires

  $$\alpha > \|\mathbf{T_G}\|_{\mathsf{GS}} \cdot \|\mathbf{R}_{\mathsf{BP}}\| \cdot \omega(\sqrt{\log m}) \tag{2.7}$$

  where $\|\mathbf{T_G}\|_{\mathsf{GS}}$ refers to the norm of Gram-Schmidt orthogonalisation of $\mathbf{T_G}$. Hence, we proceed as follows:

  1. From Lemma 2.4.6, we have that $\|\mathbf{R}_{\mathsf{BP}}\|_\infty = O(m \cdot L)$.
  2. We then get $\|\mathbf{R}_{\mathsf{BP}}\|$ as follows:

     $$\|\mathbf{R}_{\mathsf{BP}}\| := \sup_{\mathbf{x} \in S^{m-1}} \|\mathbf{R}_{\mathsf{BP}} \cdot \mathbf{x}\| \leq m \cdot \|\mathbf{R}_{\mathsf{BP}}\|_\infty \leq O(m^2 \cdot L)$$

  3. Finally, we substitute this value in Equation 2.7 to get the value of α required for the $\mathsf{SampleRight}$ algorithm.

     $$\alpha \geq O(m^2 \cdot L) \cdot \omega(\sqrt{\log m}) \tag{2.8}$$

     with $\|\mathbf{T_G}\|_{\mathsf{GS}}$ being a constant.

  The value of the parameter α required for the $\mathsf{SampleLeft}$ algorithm (Algorithm 2.3) is

  $$\alpha \geq \|\mathbf{T_A}\|_{\mathsf{GS}} \cdot \omega(\sqrt{\log 2m}) \geq O(\sqrt{n \log q}) \cdot \omega(\sqrt{\log 2m}) \tag{2.9}$$

  Thus, to satisfy both Equation 2.8 and Equation 2.9, we set the parameter

  $$\alpha \geq O(m^2 \cdot L) \cdot \omega(\sqrt{\log m}) = \tilde{O}(n \log q)^2 \cdot L$$

  Thus, the outputs of the $\mathsf{SampleLeft}$ and the $\mathsf{SampleRight}$ algorithms will be statistically indistinguishable from each other, when provided with the same set of inputs $\mathbf{F}$ and $\mathbf{u}$.

When our scheme is instantiated with these parameters, the correctness (see Lemma 2.5.1) of the scheme is satisfied when

$$O((n \log q)^6 \cdot L^2 \cdot B) < q/4$$

Clearly, this condition is satisfied when $q = \tilde{O}(n^7 L^2)$. Also, this value of $q = \text{poly}(n)$ (for any $L = \text{poly}(n)$), enables both the quantum reduction [186] and the classical reduction [180] from $\mathsf{dLWE}_{n,q,\chi}$ to approximating lattice problems in the worst case, when n, χ chosen as described above. To conclude this section, for a given max length L and an LWE dimension $n = n(\lambda)$, we set the parameters for our scheme to satisfy both the correctness and security, as follows:

$$\begin{aligned} \chi &= D_{\mathbb{Z},\sqrt{n}} \\ B &= O(n) \\ q &= \tilde{O}(n^7 L^2) \\ m &= O(n \log q) \\ s &= O(n \log q) \\ \alpha &= \tilde{O}(n \log q)^2 \cdot L \end{aligned}$$

2.7 Single-key Functional Encryption with short secret keys

Now, we use our ABE scheme to construct a single-key secure FE scheme (SKFE) with short secret keys. We will use the transformation provided by Goldwasser et al. [104] who used ABE and fully-homomorphic encryption to construct single-key functional encryption and reusable garbled circuits.

2.7.1 Definitions

We recall the functional encryption definition from the literature [141, 44, 106] with some notational changes.

A functional encryption scheme $\mathcal{FE}$ for a class of predicates with an n-bit input $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ is a tuple of four p.p.t. algorithms ($\mathsf{FE.Setup}$, $\mathsf{FE.Keygen}$, $\mathsf{FE.Enc}$, $\mathsf{FE.Dec}$) such that:

$\mathsf{FE.Setup}(1^\lambda) \rightarrow (\mathsf{mpk}, \mathsf{msk})$: The setup algorithm takes as input the security parameter 1^λ and outputs a master public key mpk and a master secret key msk.

$\mathsf{FE.Keygen}(\mathsf{msk}, P) \rightarrow \mathsf{sk}_P$: The key generation algorithm takes as input the master secret key msk and a predicate $P \in \mathcal{P}$ and outputs a key sk_P.

$\mathsf{FE.Enc}(\mathsf{mpk}, x) \rightarrow \mathsf{ct}_x$: The encryption algorithm takes as input the master public key mpk and an input $x \in \{0,1\}^*$ and outputs a ciphertext ct_x.

$\mathsf{FE.Dec}(\mathsf{sk}_P, \mathsf{ct}_x)$: The decryption algorithm takes as input a key sk_P and a ciphertext ct_x and outputs a value y.

Definition 2.7.1 (Correctness)**.** *For any polynomial $n(\cdot)$, for every sufficiently large security parameter λ, for $n = n(\lambda)$, for all $P \in \mathcal{P}_n$, and all $x \in \{0,1\}^n$,*

$$\Pr[(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda); \mathsf{sk}_P \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, P); \mathsf{ct}_x \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x) : \\ \mathsf{FE.Dec}(\mathsf{sk}_P, \mathsf{ct}_x) = P(x)] = 1 - \mathrm{negl}(\lambda).$$

Security of Single Key Functional Encryption

Intuitively, the security of SKFE requires that an adversary should not learn anything about the input x other than the computation result $P(x)$, for some predicate P for which a key was issued (the adversary can learn the predicate description P). Two notions of security have been used in the previous works: full and selective security, with the same meaning as for ABE. We present both definitions because we achieve them with different parameters of the gapSVP assumption. Our definitions are simulation-based: the security definition states that whatever information an adversary is able to learn from the ciphertext and the function keys can be simulated given only the function keys and the output of the function on the inputs.

Definition 2.7.2 (FULL-SIM- FE Security)**.** *Let $\mathcal{FE}$ be a functional encryption scheme for the family of predicates $\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$. For every p.p.t. adversary $A = (A_1, A_2)$ and p.p.t. simulator S, consider the following two experiments:*

$\underline{\exp^{\mathsf{real}}_{\mathcal{FE},A}(1^\lambda)}$:	$\underline{\exp^{\mathsf{ideal}}_{\mathcal{FE},A,S}(1^\lambda)}$:
1: $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$	
2: $(P, \mathsf{state}_A) \leftarrow A_1(\mathsf{mpk})$	
3: $\mathsf{sk}_P \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, P)$	
4: $(x, \mathsf{state}'_A) \leftarrow A_2(\mathsf{state}_A, \mathsf{sk}_P)$	
5: $\mathsf{ct}_x \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x)$	*5:* $\tilde{\mathsf{ct}}_x \leftarrow S(\mathsf{mpk}, \mathsf{sk}_P, P, P(x), 1^{\lvert x\rvert})$
6: Output $(\mathsf{state}'_A, \mathsf{ct}_x)$	*6:* Output $(\mathsf{state}'_A, \tilde{\mathsf{ct}}_x)$

The scheme is said to be (single-key) FULL-SIM*−secure if there exists a p.p.t. simulator S such that for all pairs of p.p.t. adversaries (A_1, A_2), the outcomes of the two experiments are computationally indistinguishable:*

$$\left\{ \exp^{\mathsf{real}}_{\mathcal{FE},A}(1^\lambda) \right\}_{\lambda\in\mathbb{N}} \overset{c}{\approx} \left\{ \exp^{\mathsf{ideal}}_{\mathcal{FE},A,S}(1^\lambda) \right\}_{\lambda\in\mathbb{N}}$$

We now define selective security, which is a weakening of full security, by requiring the adversary to provide the challenge input x before seeing the public key or any other information besides the security parameter. We simply specify the difference from full security.

Definition 2.7.3 (SEL-SIM-FE Security)**.** *The same as Def. 2.7.2, but modify the game so that the first step consists of A specifying the challenge input x given only the security parameter.*

2.7.2 Construction

In this section we show the gain in efficiency in the size of the secret key for the SKFE scheme that we obtain by using our ABE schemes.

From [104], we know how to obtain a single-key functional encryption scheme for a class of predicates $\mathcal{P}$ from:

1. Fully-homomorphic encryption for $\mathcal{P}$
2. Attribute-based encryption supporting a predicate class that includes FHE evaluation algorithms for $\mathcal{P}$

3. One-time garbled circuits (Eg. Yao's garbled circuits [210])

Theorem 2.7.1 ([104]). *There is a (fully/selectively secure) single-key functional encryption scheme $\mathcal{FE}$ for any class of predicates $\mathcal{P}$ that take ℓ bits of input and produce a one-bit output, assuming the existence of (1) $\mathcal{P}$-homomorphic encryption scheme, (2) a (fully/selectively) secure ABE scheme for a related class of predicates and (3) Garbling Scheme, where:*

1. *The size of the secret key is $2 \cdot \alpha \cdot \mathsf{abe.keysize}$, where $\mathsf{abe.keysize}$ is the size of the ABE key for circuit performing homomorphic evaluation of C and outputting a bit of the resulting ciphertext.*

2. *The size of the ciphertext is $2 \cdot \alpha \cdot \mathsf{abe.ctsize}(\ell \cdot \alpha + \gamma) + \mathrm{poly}(\lambda, \alpha, \beta)$*

where (α, β, γ) denote the sizes of the FHE (ciphertext, secret key, public key), respectively. $\mathsf{abe.keysize}, \mathsf{abe.ctsize}(k)$ are the size of ABE secret key, ciphertext on k-bit attribute vector and λ is the security parameter.

We use the same transformation by replacing the ABE scheme of [107] with our ABE scheme. The FHE of [51] can also be instantiated assuming the polynomial hardness of LWE.

Corollary 2.7.2. *Combining our ABE construction (Lemma 2.5.2 and Section 2.5) with Theorem-2.7.1 and assuming a NC^1 representation of the evaluation algorithm of [51] when evaluating NC^1 predicates, we obtain a single-key functional encryption scheme for a family of length-$L = \mathrm{poly}(\lambda)$ branching programs based on the polynomial hardness of LWE with the size of the secret keys being $L + \mathrm{poly}(\lambda, \log L)$, where λ is the security parameter.*

2.8 Conclusion and next steps

This chapter provided an attribute-based encryption scheme and a single-key functional encryption scheme supporting the computation class of branching programs. The asymptotic parameters for the instantiation of our ABE scheme were provided.

Good news for ABE Over the last few years, there has been an extensive work in the implementation of lattice-based cryptographic algorithms. The primitive lattice-based algorithms like key-exchange and signatures are interesting for their security against

quantum adversaries and the NIST competition [171]. The advanced lattice-based algorithms like fully-homomorphic encryption are interesting for their potential application in outsourced computations. The progress made here is not limited to these applications. The tools developed also help in the implementation of other lattice-based cryptographic primitives like ABE. There are real-world applications like key management in Huawei devices that use ABE with a simple class of functionalities. Some properties like short secret keys are currently available only in lattice-based ABE constructions. Hence, there is huge potential for lattice-based ABE constructions to find real-world applications once the core lattice algorithms become practical.

But for FE... The lattice-based construction for functional encryption though requires more optimizations than the ABE scheme to achieve practical performance. As we will discuss in later chapters, the FE constructions like ours that are based on standard cryptographic assumptions have their parameters grow with the number of computation secret keys supported [106], even if we were to waive the need for short secret keys. The performance overhead worsens when supporting more expressive functionalities required for real world applications. This motivates the need for new perspectives for designing FE.

Chapter 3

A Formal Introduction to SGX

3.1 Intel SGX Background

Intel Software Guard Extensions (SGX) [158] is a set of processor extensions to Intel's x86 design that allow for the creation of isolated execution environments called enclaves. These isolated execution environments are designed to run software and handle secrets in a trustworthy manner, even on a host where all the system software (including OS, hypervisor, etc) and system memory are untrusted. The isolation of enclave resident applications from all other processes is enforced by hardware access controls. The SGX specifications are detailed and complex [129, 158].

There are three main functionalities that enclaves achieve:

- *Isolation*–code and data inside the enclave protected memory cannot be read/modified by any process external to the enclave.
- *Sealing*–data passed to the host environment is encrypted and authenticated with a hardware-resident key.
- *Attestation*–a special signing key and instructions are used to provide an unforgeable report attesting to code, static data, and (hardware-specific) metadata of an enclave, as well as outputs of computations performed inside the enclave.

Let us explain these three functionalities in more detail.

3.1.1 Isolation.

Enclaves reside in a hardware guarded area of memory called the Enclave Page Cache (EPC). The EPC is currently limited to 128 MB, consisting of 4KB page chunks, and applications can use approximately 90 MB. When an enclave program is loaded, its code and static data are copied from untrusted memory to pages inside the EPC. A measurement of the contents of these pages called MRENCLAVE (essentially a SHA256 hash of the page contents) is also stored inside the EPC in a structure that is linked to the enclave. Entry into the enclave is not permitted throughout this process until the measurement has been finalized. The creation process establishes an enclave identity, which is used as a handle to transfer program control to the enclave. The hardware enforces that only the executable code pages associated with a particular enclave identity can access the other pages associated with that identity.

3.1.2 Sealing.

Every SGX processor has a key called the Root Seal Key that is embedded during the manufacturing process. An enclave can use the `EGETKEY` instruction to derive a key called *Seal Key* from the Root Seal Key that is specific to the enclave identity, which can be used to encrypt/authenticate data and store it in untrusted memory. Sealed data can be recovered by the same enclave even after enclave is destroyed and restarted on the same platform. But the Seal key cannot be derived by a different enclave on the same platform or any enclave on a different platform.

3.1.3 SGX Attestation

Local attestation Local attestation is between two enclaves on the same platform. The program in the enclave generating the attestation specifies a target enclave that will verify the attestation and invokes the `EREPORT` instruction. `EREPORT` first generates a *report* containing the MRENCLAVE and metadata of the calling enclave, fetching this information directly from protected memory and registers. The report may also include additional data provided by the calling enclave. Second, `EREPORT` uses the target enclave specification (measurements and metatada) to derive the target enclave's *Report Key* from the Root Seal Key. It then uses this Report Key to compute a MAC over the *report*. Any enclave

can use the `EGETKEY` instruction to fetch its own Report Key, derived from the Root Seal Key and measurements/metadata linked to the enclave. Thus, the target enclave, which resides on the same platform as the attesting enclave and shares the same Root Seal Key, will be able to derive the Report Key it needs to verify the MAC on *report*.

Remote attestation Local attestation is leveraged in remote attestation, which generates enclave reports that can be verified by remote parties. Roughly, a special enclave called the Quoting Enclave will process local attestations from other enclaves and convert these into remote attestations called *quotes*. More specifically, the Quoting Enclave possesses a private member key for an anonymous group signature scheme called Intel Enhanced Privacy ID (EPID) [135] that is uses to sign reports received from other locally attesting enclaves. In EPID, an *issuer* (in this case Intel) generates a group public key gpk, and registers members of the group by issuing member private keys. Member keys are issued through a *blind join* protocol and are unknown to the issuer. Signatures generated from private member keys can be publicly verified using gpk, but cannot be linked to any particular member key.[1]

EPID key provisioning The Quoting Enclave obtains the EPID private key through an involved process with the Intel Provisioning Server. Every SGX CPU has another embedded key called the Root Provisioning Key. Unlike the Root Seal Key, the Root Provisioning Key is also given to the Intel Provisioning Server. Another special enclave called the Provisioning Enclave calls `EGETKEY` to derive a Provisioning Key from the Root Provisioning Key incorporating specific information about the trusted computing base (TCB), enclave measurements, and metadata. Since the Intel Provisioning Server can derive the same key, the Provisioning Enclave symmetrically authenticates to the Intel Provisioning Server, demonstrating that it is a valid Provisioning Enclave running on a genuine Intel SGX CPU at a specific TCB. Finally, an EPID private member key is delivered to the Provisioning Enclave through the EPID blind join protocol, and this key is passed to the Quoting Enclave.

3.1.4 SGX TCB.

SGX stands out in that its TCB consists only of the CPU microcode and privileged containers, however it also requires the user to trust in Intel's key management infrastructure

[1] Currently, EPID signatures need to be verified by contacting the Intel Attestation Server.

for signing microcode and various service enclaves. In particular, we must trust that the root seal keys embedded into devices are not leaked from the manufacturing facility, and that the Intel Provisioning Server safely manages root provisioning keys as well as EPID master secret keys.

3.1.5 SGX side-channel attacks and defenses

Cache-timing attacks [75] cause cache misses and thus may observe enclave memory access patterns at cache-line granularity. Similarly, page-fault attacks [208] can cause enclave page lookups to result in page-faults and thus may observe enclave memory access patterns at 4KB page granularity. Next, branch shadowing may directly infer control flow (i.e branches) in an enclave process. Branch shadowing exploits the fact that SGX does not erase branch history, which is used by the CPU for branch prediction, and is important for performance of the instruction pipeline. The attack infers from timing differences in branch prediction whether a target branch is stored in the branch history. And, finally synchronization bugs in the multi-threaded code running in SGX could potentially lead to even circumventing the Intel licensing procedure in creating SGX production enclaves [207]. These bugs are relatively easier to exploit in SGX than outside because the attacker model allows an untrusted OS which can control the thread scheduling of enclaves.

One defense against all the above software attacks is to ensure that enclave programs are *data-oblivious*, i.e. do not have memory access patterns or control flow branches that depend on the values of sensitive data. Ohrimenko et. al. [173] take this approach in their design of privacy-preserving multi-party machine learning using SGX. T-SGX [194] also provides countermeasures against controlled-channel attacks. A more general approach is to use ORAM techniques, as in [185, 155], though this can result in a considerable performance overhead. Several countermeasures to the branch shadowing attack, both hardware and software based, were proposed in [146]. Hardware countermeasures would require changes to SGX architecture. Defense mechanisms against different kinds of synchronization bugs already exist as listed by [207]. SGX-Shield [193] enables ASLR for SGX, which helps defend against these attacks in general. More recently, SGX has also been shown to be vulnerable to the “microarchitectural implementation bugs” [154] in the x86 architecture [56]. As [56] explains, this is due to the tight coupling of SGX with x86 and not due to the architectural design of SGX.

Sanctum [76] is an academic SGX-like system that is resilient to both cache-timing and page-fault attacks, demonstrating that these attacks are not inherent in SGX-like

systems. SGX is an evolving technology, and so we can expect that even hardware based countermeasures could be incorporated into future SGX versions (see changes already in SGX2 [132]).

3.2 Formal Models and Definitions

In this section, we will formally define a model for trusted hardware inspired by SGX.

3.2.1 Formal HW model

We describe a black-box program HW that captures the trusted hardware's functionality and its interface exposed to the user.

Definition 3.2.1. *The functionality* HW *for a class of (probabilistic polynomial time) programs* $\mathcal{Q}$ *consists of* $\mathsf{HW.Setup}$, $\mathsf{HW.Load}$, $\mathsf{HW.Run}$, $\mathsf{HW.Run\&Report}$, $\mathsf{HW.Run\&Quote}$, $\mathsf{HW.ReportVerify}$, $\mathsf{HW.QuoteVerify}$. HW *has an internal state* state *that consists of two variables* $\mathsf{HW.sk_{quote}}$ *and* $\mathsf{HW.sk_{report}}$ *and a table* T *consisting of enclave state tuples indexed by enclave handles.*

- $\mathsf{HW.Setup}(1^\lambda)$: This takes in a security parameter λ and generates the secret keys $\mathsf{sk_{quote}}$, $\mathsf{sk_{report}}$, and stores these in $\mathsf{HW.sk_{quote}}, \mathsf{HW.sk_{report}}$ respectively. Finally, it generates and outputs public parameters params.

- $\mathsf{HW.Load}(\mathsf{params}, Q)$: This loads a stateful program into an enclave. $\mathsf{HW.Load}$ takes as input a program $Q \in \mathcal{Q}$ and some global parameters params. It first creates an enclave and loads Q and generates a handle hdl that will be used to identify the enclave running Q. It initializes the entry $T[\mathsf{hdl}] = \emptyset$ and outputs hdl.

- $\mathsf{HW.Run}(\mathsf{hdl}, \mathsf{input})$: This runs an enclave program. It takes in a handle hdl corresponding to an enclave running the stateful program Q and an input input. It runs Q at state $T[\mathsf{hdl}]$ with input input and records the output output. It sets $T[\mathsf{hdl}]$ to be the updated state of Q and outputs output.

- $\mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl}, \mathsf{input})$: This executes a program in an enclave and also generates an attestation of its output that can be verified by an enclave program on the same HW platform. It takes as inputs a handle hdl for an enclave running a program Q and an input input for Q. The algorithm first executes Q on input

to get output, and updates $T[\mathsf{hdl}]$ accordingly. $\mathsf{HW.Run\&Report}$ outputs the tuple $\mathsf{report} := \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$, where $\mathsf{md}_{\mathsf{hdl}}$ is the metadata associated with the enclave, tag_Q is a program tag that can be used to identify the program running inside the enclave (it can be a cryptographic hash of the program code Q) and mac is a cryptographic MAC produced using $\mathsf{sk}_{\mathsf{report}}$ on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output})$.

- $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}, \mathsf{input})$: This executes a program in an enclave and also generates an attestation of its output that can be publicly verified, e.g. by a remote party. This takes as inputs a handle hdl corresponding to an enclave running a program Q and an input input for Q. This algorithm has restricted access to the key $\mathsf{sk}_{\mathsf{HW}}$ for using it to sign messages. The algorithm first executes Q on input to get output, and updates $T[\mathsf{hdl}]$ accordingly. $\mathsf{HW.Run\&Quote}$ then outputs the tuple $\mathsf{quote} := \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \sigma\big)$, where $\mathsf{md}_{\mathsf{hdl}}$ is the metadata associated with the enclave, tag_Q is a program tag for Q and σ is a signature on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output})$.

- $\mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}', \mathsf{report})$: This is the report verification algorithm. It takes as inputs, a handle hdl' for an enclave and a $\mathsf{report} = \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$. It uses $\mathsf{sk}_{\mathsf{report}}$ to verify the MAC. If mac is valid, it outputs 1 and adds a tuple $(\mathsf{report}, 1)$ to $T[\mathsf{hdl}']$. Otherwise it outputs 0 and adds $(\mathsf{report}, 0)$ to $T[\mathsf{hdl}']$.

- $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$: This is the quote verification algorithm. This takes params and $\mathsf{quote} = \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \sigma\big)$ as input. It outputs 1 if the signature verification of σ succeeds. It outputs 0 otherwise.

- $\mathsf{HW.Seal}_{\mathsf{sk}'_{\mathsf{HW}}}(\mathrm{AAD}, \mathsf{msg})$: This is the sealing algorithm which is an authenticated encryption algorithm. This takes a message and an additional data AAD as inputs and outputs the ciphertext $\mathsf{seal_ct}$.[2] Here, AAD is the *additional authentication data* which is included as a part of the MAC step to provide integrity but not encrypted along with msg. We will ignore the AAD argument when there is none.

- $\mathsf{HW.Unseal}_{\mathsf{sk}'_{\mathsf{HW}}}(\mathrm{AAD}, \mathsf{seal_ct})$: This is the unsealing algorithm which is the decryption for the authenticated encryption. This takes the ciphertext $\mathsf{seal_ct}$ and the additional data AAD as inputs and outputs msg or $\perp$.

In Section 3.3, we formally define the correctness of HW as well as the security properties of $\mathsf{HW.Run\&Report}$, $\mathsf{HW.Run\&Quote}$, $\mathsf{HW.ReportVerify}$, and $\mathsf{HW.QuoteVerify}$ as local attestation unforgeability (LocAttUnf) and remote attestation unforgeability (RemAttUnf).

[2] SGX uses AES-GCM to encrypt msg using the Seal key of the enclave calling the function.

Oracles and handles HW models a single SGX chip. When a system involves multiple HW platforms, each is modeled by a separate HW instance. When a particular process needs to interact with multiple platforms, the remote interactions are modeled through oracle calls, which in the real world corresponds to communicating with a process running on the relevant remote machine. The handles in the model generated by HW.Load do not need to secret or unpredictable. They are only relevant to the interfaces described in HW, which by definition can only be accessed by the HW instance itself. More concretely, in the real world SGX instantiation, these enclave handles are used only by processes running on the same machine as the enclave(s).

Modeling assumptions One way of viewing this definition of HW is that it describes the ideal functionality or oracle that models the real (physical) world assumptions about the hardware security properties of Intel SGX), and that an adversary shouldn't be able to distinguish between interacting with the real world hardware and the ideal functionality. This allows us to simulate the adversary's interaction with HW in a proof of security, but it is a very strong assumption on the trusted hardware being used, particularly since the adversary has access to the physical hardware and can closely monitor its behavior. A weaker assumption, stated informally, is simply that the adversary gains no more "useful" information from querying the real hardware on some input beyond the outputs specified by HW, without requiring that an adversary's physical interactions with HW cannot be simulated. We explore both models in our construction of functional encryption from HW, though it turns out that we cannot achieve the standard non-interactive notion of functional encryption in the stronger security model.

Related models Barbosa et. al. [29] define a similar interface/ideal functionality to represent systems like SGX that perform attested computation. Compared to their model, our model sacrifices some generality for a simpler syntax that more closely models SGX. Their security model uses a game-based definition of attested computation, similar to the second security model we discuss in Definition 4.6.2.

Pass, Shi, and Tramer [179] also define an ideal functionality for attested computation in the Universal Composability framework [59]. The goal of their model is to explore *composable* security for protocols using secure processors performing attested computation. Similar to [29] their syntax is more abstract that ours, e.g. does not distinguish between local and remote attestation. However, their hardware security model is more similar in that it allows the hardware functionality to be simulated. A key difference is that their simulator does not possess the hardware's secret signing key(s) used to generate

attestations. Our simulator will be given the hardware's secret keys, similar to trapdoor information in CRS-model proofs.

Bahmani et al [25] adapts the SGX model of [29] to deal with sequences of SGX computations that may be stateful, asynchronous, and interleaved with other computations. Their model is called *labelled attested computation*, which refers to labels being appended to every enclave input/output in order to track state. This capability is implicitly captured in our model as well.

3.3 HW correctness and security definitions

Correctness A HW scheme is correct if the following things hold (using the syntax from Definition 3.2.1): For all $Q \in \mathcal{Q}$, all input in the input domain of Q and all handles $\mathsf{hdl}' \in \mathcal{H}$,

- Correctness of Run: $\mathsf{output} = Q(\mathsf{input})$ if Q is deterministic. More generally, $\exists$ random coins r (sampled in run time and used by Q) such that $\mathsf{output} = Q(\mathsf{input})$.
- Correctness of Report and $\mathsf{ReportVerify}$:

$$\Pr\left[\mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}', \mathsf{report}) = 0\right] = \text{negl}(\lambda)$$

- Correctness of Quote and $\mathsf{QuoteVerify}$:

$$\Pr\left[\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote}) = 0\right] = \text{negl}(\lambda)$$

We now define the security properties of the HW primitive.

3.3.1 Local attestation unforgeability

The local attestation unforgeability (LocAttUnf) security is defined similarly to the unforgeability security of a MAC scheme. Informally, it says that no adversary can produce a $\mathsf{report} = \left(\mathsf{md}'_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\right)$ that verifies correctly for any $\mathsf{hdl}' \in \mathcal{H}$ and $\mathsf{output} = Q(\mathsf{input})$, without querying the inputs $(\mathsf{hdl}, \mathsf{input})$.

This is formally defined by the following security game.

Definition 3.3.1. (LocAttUnf-HW). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *$\mathcal{C}$ runs the* $\mathsf{HW.Setup}(1^\lambda)$ *algorithm to obtain the public parameters* params*, secret keys* $(\mathsf{sk_{HW}}, \mathsf{sk_{report}})$ *and an initialization string* state*. It gives* params *to* $\mathcal{A}$*, and keeps* $(\mathsf{sk_{HW}}, \mathsf{sk_{report}})$ *and* state *secret in the trusted hardware.*

2. *$\mathcal{C}$ initializes a list* $\mathsf{query} = \{\}$.

3. *$\mathcal{A}$ can run* $\mathsf{HW.Load}$ *on any input* (params, Q) *of its choice and get back* hdl.

4. *$\mathcal{A}$ can run* $\mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}$ *on input* $(\mathsf{hdl}, \mathsf{input})$ *of its choice and get* $\mathsf{report} := \big(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$*. For every run,* $\mathcal{C}$ *adds the tuple* $(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output})$ *to the list* query.

5. *$\mathcal{A}$ can also run* $\mathsf{HW.ReportVerify}$ *on input* $(\mathsf{hdl}', \mathsf{report})$ *of its choice and gets back the result.*

We say the adversary wins the above experiment if:

1. $\mathsf{HW.ReportVerify}_{\mathsf{sk_{report}}}(\mathsf{hdl}'^*, \mathsf{report}^*) = 1$*, where* $\mathsf{report}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{input}^*, \mathsf{output}^*, \mathsf{mac}^*)$ *and*

2. $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{input}^*, \mathsf{output}^*, \mathsf{mac}^*)$ *was not added to* query *before* $\mathcal{A}$ *queried* $\mathsf{HW.ReportVerify}$ *on* $(\mathsf{hdl}'^*, \mathsf{report}^*)$.

The HW *scheme is LocAttUnf-HW secure if no adversary can win the above game with non-negligible probability.*

3.3.2 Remote attestation unforgeability

The remote attestation unforgeability (RemAttUnf) security is defined similarly to the unforgeability security of a signature scheme. Informally, it says that no adversary can produce a $\mathsf{quote} = \big(\mathsf{hdl}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \pi\big)$ that verifies correctly and $\mathsf{output} = Q(\mathsf{input})$, without querying the inputs $(\mathsf{hdl}, \mathsf{input})$.

This is formally defined by the following security game.

Definition 3.3.2. (RemAttUnf-HW). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *$\mathcal{C}$ runs the* $\mathsf{HW.Setup}(1^\lambda)$ *algorithm to obtain the public parameters* params*, secret keys* $(\mathsf{sk_{HW}}, \mathsf{sk_{report}})$ *and an initialization string* state*. It gives* params *to* $\mathcal{A}$*, and keeps* $(\mathsf{sk_{HW}}, \mathsf{sk_{report}})$ *and* state *secret in the trusted hardware.*

2. *$\mathcal{C}$ initializes a list* $\mathsf{query} = \{\}$.

3. *$\mathcal{A}$ can run* $\mathsf{HW.Load}$ *on any input* (params, Q) *of its choice and get back* hdl.

4. *Also, $\mathcal{A}$ can run* $\mathsf{HW.Run\&Quote}$ *on input* $(\mathsf{hdl}, \mathsf{input})$ *of its choice and get* $\mathsf{quote} := (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \pi)$. *For every run, $\mathcal{C}$ adds the tuple* $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output})$ *to the list* query.

5. *Finally, the adversary outputs* $\mathsf{quote}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{input}^*, \mathsf{output}^*, \pi^*)$.

We say the adversary wins the above experiment if:

1. $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote}^*) = 1$,

2. $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{input}^*, \mathsf{output}^*) \notin \mathsf{query}$

The HW *scheme is RemAttUnf-HW secure if no adversary can win the above game with non-negligible probability.*

Note that the scheme is secure even if $\mathcal{A}$ can produce a quote^* different from the query outputs for some $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{input}^*, \mathsf{output}^*) \in \mathsf{query}$. But quote^* cannot be a proof for a different program or input or output. This definition resembles the existential unforgeability like notions.

The security of the sealing procedure using Seal and Unseal is the same as the security of an authenticated encryption scheme [188, 34].

We also point out some other important properties of the trusted hardware that we impose in our model.

- Any user only has black box access to these algorithms and hence hidden from the internal secret key $\mathsf{sk}_{\mathsf{HW}}$, initial state state or intermediary states of the programs running inside secure containers.

- The output of the $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}$ algorithm is succinct: it does not include the full program description, for instance.

3.4 Differences between HW and Intel SGX

Here we list and justify the simplifications we made to SGX in order to formally model its functionality and reason about the security of our system.

- In our model, HW is a black-box program that loads and manages enclaves, which includes updating their state in HW.ReportVerify. This internal management is entirely hidden from the user, which only sees the interface, inputs, and outputs. In real Intel SGX, only operations internal to a program running in an enclave (i.e. instructions that operate on registers/memory in the EPC) are entirely hidden from the user, and the enclave program's state cannot be modified by an external entity. Programs running in enclaves can directly run instructions to generate and verify reports.

- The key $\mathsf{sk}_{\mathsf{HW}}$ used to sign remote attestations in our model is generated during HW.Setup (i.e. in the trusted manufacturing facility). In Intel SGX, this key is not actually fused into the device. It is delivered to an special enclave QE (the Quoting Enclave) running on the device that symmetrically authenticates to the Intel Provisioning Server by accessing a key (the Root Provisioning Key) that is fused into the device and also given to Intel. The QE then receives the private key for a group signature scheme through a blind join protocol (see [135]), and uses this key to generate quotes on behalf of other enclaves. Our model compresses the manufacturing and provisioning processes into HW.Setup.

- Our HW.Run&Report algorithm generates a report that can be verified by any enclave on the same HW platform. In SGX, a report is generated for a specific destination enclave and only that enclave can verify its validity. However, this particular feature of SGX is not relevant for our application.

- Intel SGX also has the capability of sealing data with a hardware fused Seal Key. In particular, this allows the device to use persistent storage for keys. For simplicity, we do not include this in our formal model, and assume the trusted HW functionality is persistent.

- HW.Run&Quote and HW.QuoteVerify use a standard cryptographic signature scheme to sign and verify quotes. In Intel SGX the signatures used for quotes are actually anonymous group signatures, but this additional property is not relevant to our application, so we omit it for simplicity. Moreover, currently Intel SGX requires the user to contact the Intel Attestation Server (IAS) to verify group signatures.

Theoretically, verifying an anonymous group signature only requires the public group key, and needn't involve the IAS.

Chapter 4

Functional Encryption from SGX

4.1 The need for practical solutions for Functional Encryption

We earlier motivated the primitive of functional encryption with a user of cloud computing using this primitive of FE to generate the parameters and the keys for the system. She ran all the $\mathsf{FE.Setup}$, $\mathsf{FE.Keygen}$ and $\mathsf{FE.Enc}$. This version of FE is called private-key functional encryption.

A public-key functional encryption is more general where a trusted authority runs $\mathsf{FE.Setup}$ to generate msk and mpk. The authority holding a master secret key msk can generate special functional secret keys, where each functional key sk_f is associated with a function (or program) f on plaintext data. When the key sk_f is used to decrypt a ciphertext ct, which is the encryption of some message m, the result is the quantity $f(m)$. Nothing else about m is revealed.

Further, Multi-Input Functional Encryption (MIFE) [103] is an extension of FE, where the functional secret key sk_g is associated with a function g that takes $\ell \geq 1$ plaintext inputs. When invoking the decryption algorithm D on inputs $D(\mathsf{sk}_g, c_1, \ldots, c_\ell)$, where ciphertext number i is an encryption of message m_i, the algorithm outputs $g(m_1, \ldots, m_\ell)$. Again, nothing else is revealed about the plaintext data $m_1, \ldots, m_\ell$. Functions can be deterministic or randomized with respect to the input in both single and multi-input settings [112, 103]. If FE and MIFE could be made practical, they would have numerous real-world applications.

The problem is that currently there aren't any practical constructions of FE from standard cryptographic assumptions for anything more than simple functionalities (e.g., inner products). Moreover, there is evidence that constructing general-purpose FE is as hard as constructing program obfuscation [93, 35, 19]. However, existing candidate constructions for obfuscation are impractical [147] and rely on very new and unestablished computational hardness assumptions, some of which have been broken [165, 70]. Previous work proposed using trusted hardware to instantiate FE, however it relied on simulatable hardware "tokens" which did not model real hardware [72].

4.2 Our contributions

We propose the first *practical and provably secure FE system that can be instantiated today from real commonly available hardware.* We implemented our proposed system, called IRON, using Intel's Software Guard Extensions (SGX) and performed evaluation to show its practical efficiency compared with alternative cryptographic algorithms. We also propose a formal cryptographic model for analyzing the security of an SGX-based FE system and prove that IRON satisifes our security definitions.

Intel SGX provides hardware support for isolated program execution environments called *enclaves*. Enclaves are encrypted memory containers that protect against operating system, hypervisor, physical, and malware attacks. However, designing a *provably secure application* from Intel SGX is a non-trivial task. While a number of works showed how to build cryptographic algorithms and systems from Intel SGX [190, 21, 201, 195, 211, 122, 29, 25], only a handful of works have attempted to model and prove systems security from Intel SGX [122, 29, 25, 179]. Reminiscent to secure protocols (such as SSL/TLS), which are easy to construct from basic cryptographic primitives, but are notoriously hard to analyze and prove, doing so requires careful understanding of nuances and techniques. We believe Intel SGX (and similar trusted hardware technologies) will become standard cryptographic tools for building secure systems. Thus, it is important to understand how to build a system with a formal model and guarantees from the beginning.

Establishing a rigorous connection between IRON and the cryptographic notion of FE is also particularly useful since FE is a very general and powerful primitive that can be used to directly construct many other cryptographic primitives, including fully homomorphic encryption (FHE) [60, 16] and obfuscation [35, 19]. Thus, rather than a complete system on its own, we view IRON as a basic framework upon which a family of more application-specific systems can be built in the future, and automatically inherit IRON's rigorous notion of security.

The security of IRON relies on trust in Intel's manufacturing process and the robustness of the SGX system. While we focus on implementing IRON with Intel SGX, in principle the system could be instantiated using other isolated execution environments that also support remote software attestation, such as XOM [152], AEGIS [199, 200], Bastion [65], Ascend [88] and Sanctum [76]. Each of these systems have slightly different trust assumptions and trusted computing bases (TCBs). A detailed comparison of these systems to Intel SGX is covered in [75]. It is important to acknowledge the limitations of basing security on trust in any particular hardware design. For instance, several side-channel attacks have come to light since SGX's initial release [208, 207, 146, 52, 192]. In our system, we ensure that the functionalities we implemented are resistant to known side-channel attacks on SGX. Generic techniques for protection against enclave side channels are also under study in various works [185, 155, 207, 146, 193].

4.2.1 Construction overview

The design of IRON is described in detail in Section 4.4. At a high level, the system uses a *Key Manager Enclave* (KME) that plays the role of the trusted authority who holds the master key. This authority sets up a standard public key encryption system and signature scheme. Anyone can encrypt data using the KME's published public key. When a client (e.g., researcher) wishes to run a particular function f on the data, the client requests authorization from the KME. If approved, the KME releases a functional secret key sk_f that takes the form of an ECDSA signature on the code of f. Then, to perform the decryption, the client runs a *Decryption Enclave* (DE) running on an Intel SGX platform. Leveraging remote attestation, the DE can obtain over a secure channel the secret decryption key from the KME to decrypt ciphertexts. The client then loads sk_f into the DE, as well as the ciphertext to be operated on. The DE, upon receiving sk_f and a ciphertext, checks the signature on f, decrypts the given ciphertext, and outputs the function f applied to the plaintext.

We implemented IRON and report on its performance for a number of functionalities. For complex functionalities, this implementation is (unsurprisingly) far superior to any cryptographic implementation of FE (which does not rely on hardware assumptions). We show in Section 4.5 that even for simple functionalities, such as comparison and small logical circuits, our implementation outperforms the best cryptographic schemes by over a 10,000 fold improvement. Furthermore, we discuss how IRON could support more expressive function authorization policies that are not possible with standard FE.

Security analysis. In this work we formalize our trust assumptions and definition of security for hardware-assisted FE, as well as rigorously prove the security of our system in this formal model (Section 4.6.3 and Section 4.7). While our construction of SGX-assisted FE/MIFE is clean and simple, formally proving security turns out to be complicated and non-trivial. For instance, we encounter a TLS-like situation where we have to show that no information is revealed from an encryption of m whose corresponding secret decryption key is transferred from KME to DE to the third enclave using the secure channels established between these enclaves. With an adversary being able to tamper with the inputs and the outputs of these enclaves, the "simulator" that we construct to prove the simulation-security of FE requires more care. Section 4.7 has more details on this.

4.3 Related Work

A number of papers use SGX to build secure systems. Haven [32] protects unmodified Windows applications from malicious OS by running them in SGX enclaves. Scone [21] and Panoply [195] build secure Linux containers using SGX. VC3 [190] enables secure MapReduce computations while keeping both the code and the data secret using SGX. A complete security analysis of the system was also presented but the system evaluation was performed using their own SGX performance model based on the Intel whitepapers. Ohrimenko et al. [173] present data-oblivious algorithms for some popular machine learning algorithms. These algorithms can be used in conjunction with our system if one wants an FE scheme supporting machine learning functionalities. Gupta et al. [122] proposed protocols and theoretical estimates for performing secure two-party computation using SGX based on the SGX specifications provided in Intel whitepapers. Concurrent to our work, Bahmani et al. [25] proposed a secure multi-party computation protocol where one of the parties has access to SGX hardware and performs the bulk of the computation. They evaluate their protocol for Hamming distance, Private Set Intersection and AES. This work and [179] also attempt formal modeling of SGX like we do. We discuss the comparison between the models in Section 3.2.1. Also concurrent to our work, Nayak et al. [170] designed and implemented a construction for virtual black-box obfuscation (a crypto primitive even stronger than FE) using a version of trusted hardware that they design and prototype in an FPGA. In contrast, our work focuses on studying the provable guarantees from a commercially available hardware.

[72] first proposed a way to bypass the impossibility results in functional encryption by the use of "hardware tokens". But, their work is purely theoretical and they model trusted hardware as a single stateless deterministic token, which does not capture how SGX works

because their hardware token is initialized during FE.Setup (refer Definition 5 of [72]). But in SGX, and hence in our model, the trusted hardware HW is setup and initialized *independent* of FE.Setup by the trusted hardware manufacturer, Intel. After this point, an adversary who is in possession of the hardware can monitor and tamper with all the input coming in to the hardware and the corresponding outputs. Naveed et al. [168] propose a related notion of FE called "controlled functional encryption". The main motivation of C-FE is to introduce an additional level of access control, where the authority mediates every decryption request.

In general, various forms of trusted hardware (real ones like TPM [115] and Intel TXT [128] and theoretical ones like tamper-proof tokens [139, 102]) have enabled applications like one-time programs [102], a contractual anonymity system [191], secure multi-party computation with some strong security guarantees [111] that are either not possible or not practical otherwise.

4.4 System Design

4.4.1 Architecture overview

Platforms The Iron system consists of a single *trusted authority* (Authority) platform and arbitrarily many *decryption node* platforms, which may be added dynamically. Both the trusted authority and decryption node platforms are Intel SGX enabled. Just as in a standard FE system, the Authority has the role of setting up public parameters as well as distributing *functional secret keys*, or the credentials required to decrypt functions of ciphertexts. A *client application*, which does not need to run on an Intel SGX enabled platform, will interact once with the Authority in order to obtain credentials (i.e., a secret key) for a function and will then interact with any decryption node in order to perform functional decryptions of ciphertexts.

Protocol flow The public parameters that the Authority generates includes a public encryption key for a public key cryptosystem and a public verification key for a cryptographic signature scheme. The Authority manages the corresponding secret decryption key and secret signing key. Through remote attestation, the Authority platform provisions the secret decryption key to a special enclave called a *decryption enclave* (DE) on the decryption node(s). Ciphertexts are encrypted using the public encryption key. To authorize a client application to run a function on ciphertexts, the Authority signs the function code using

its secret signing key, and sends this signature to the client. When the client sends a ciphertext, function code, and valid signature on the function code to the decryption node, the DE with access to the secret key checks the signature, decrypt the ciphertext, run the function on the plaintext, and output the result. The enclave aborts on invalid signatures.

Decryption enclaves & function enclaves Thus far in our simple description of the protocol flow, there is a single enclave on the decryption node (the DE) that manages the secret decryption key, checks function signatures, and performs functional decryption. This requires the DE to receive code as input (after enclave initialization) and to both check a signature on the code as well as execute the code. However, in the current version of SGX, enclaves cannot dynamically allocate new code pages. All enclave memory as well as the Read, Write, and Execute (RWX) permissions of each page must be committed before initialization (i.e., at build time). Therefore, the only way for the DE to execute the function it receives as native code would be to pre-allocate empty pages at build time that are both writeable and executable, and to write the function code it receives to these pages.[1] There are several drawbacks to this approach, namely that it requires the DE to predetermine the maximum size of any function it will support, and conflicts with executable space protection (the function code is more vulnerable to exploits that might overwrite code pages). A second option is to execute the function inside the DE as interpreted code, but this could greatly impact performance for more complex functions.

The third option is to load functions in entirely separate *function enclaves* and take advantage of *local attestation*, which already provides a way for one enclave to verify the code running in another. This is the cleanest design and the simplest to implement. One tradeoff, however, is that creating a new enclave for each authorized function is a relatively expensive operation. This has little impact on applications that run a few functions on many ciphertexts, but would impact applications that run many functions on only a few ciphertexts. We demonstrate in our evaluation (Section 4.5) that for a simple functionality like Identity Based Encryption (IBE) interpreting the function (i.e. identity match) in an enclave is an order of magnitude faster.

Authorization policies The Authority has full responsibility over implementing a given function authorization policy, which governs how it decides whether or not to provide a given client with a signed function. The enclaves on the decryption platform do not play any role in implementing this policy. Typically, the details of the authorization policy

[1]This will change in SGX2[132], which adds instructions to dynamically load new code pages into enclaves. We can revisit the design based on this new feature when SGX2 becomes available.

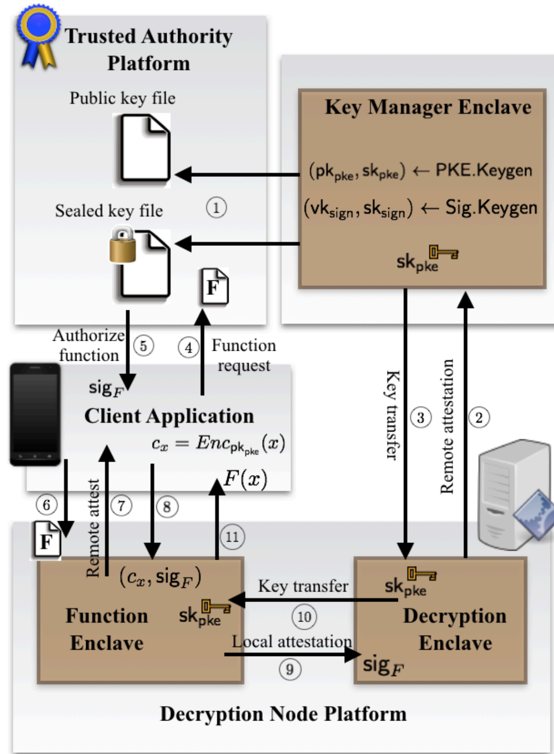


Figure 4.1: Iron Architecture and Protocol Flow

are beyond the scope of an FE construction and are application specific (we mentioned several examples in the introduction). It is important to note that in classical FE once a client obtains a secret key it can use it arbitrarily. Thus authorization policies are one-time decisions, and cannot cover key revocation, or limits on the number of times a client may run a function, etc. In contrast, more expressive policies may be possible in our SGX-assisted version of FE. For example, the secret key could be tagged with an expiration time that the enclaves on the decryption platform could check before running decryption by utilizing SGX's trusted time service [131]. Enforcing limits on the number of times a client can run a function would require maintaining non-volatile enclave state, for which SGX does not immediately provide rollback protection (see [157] for a recent system providing rollback protection using SGX's monotonic counters [130]. Additionally, it would require sharing state across all active decryption enclaves with assistance from the Authority.

Key manager enclave The Authority uses the *key manager enclave* (KME) to generate encryption and signing keys, and uses this enclave as an oracle to authorize functions. This might seem unnecessary (in our current implementation) as the Authority can use the KME to sign any function of its choice, however it offers several advantages. First, it serves as a way to protect the FE master key against an attacker that does not have long term access to the machine running the key manager enclave. Furthermore, we can imagine a more general scenario where the authorization policy is run entirely inside a key manager enclave, which only signs functions when provided with suitable proof of authorization which could come from a decentralized authority like a public blockchain or rely on an independent PKI.

4.4.2 FE Protocols

FE Setup The Authority platform runs a secure enclave called the *key manager enclave* (KME) that it uses to generate a public/private key pair $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}})$ for a CCA2 secure public key cryptosystem and a verification/signing key pair $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}})$ for a cryptographic signature scheme. The keys $\mathsf{pk}_{\mathsf{pke}}$ and $\mathsf{vk}_{\mathsf{sign}}$ are published while the keys $\mathsf{sk}_{\mathsf{pke}}$ and $\mathsf{sk}_{\mathsf{sign}}$ are sealed with the KME's sealing key and kept in non-volatile storage. Note that the Authority has full access to the KME and can thus use it to authorize any function, thus the KME is simply used for key management. The handle to the KME's signing function call, which produces signatures using $\mathsf{sk}_{\mathsf{sign}}$, serves as the trusted authority's master secret key.

FE Decrypt Setup When a new decryption node is initialized, the KME establishes a secure channel with a *decryption enclave* (DE) running on the decryption node SGX-enabled platform. The KME receives from the decryption node a *remote attestation*, which demonstrates that the decryption node is running the expected DE software and that the DE has the correct signature verification key $\mathsf{vk}_{\mathsf{sign}}$. The remote attestation also establishes a secure channel, i.e. contains a public key generated inside the DE. After verifying the remote attestation, the KME sends $\mathsf{sk}_{\mathsf{pke}}$ to the DE over the established secure channel, and authenticates this message by signing it with $\mathsf{sk}_{\mathsf{sign}}$.At this point, it is not at all obvious why the KME needs to sign its message to the DE. Indeed, since $\mathsf{sk}_{\mathsf{pke}}$ is encrypted, it seems that there isn't anything a man-in-the-middle attacker could do to harm security. If the message from the KME to the DE is replaced, the decryption node platform would simply fail to decrypt ciphertexts encrypted under $\mathsf{pk}_{\mathsf{pke}}$. However, it turns out that we have to authenticate the KME's messages for our formal proof of security to work (see Section 5.6).

FE.Keygen A client application requests from the $\mathsf{Authority}$ the "secret key" for a function f.The $\mathsf{Authority}$ decides whether the client application is authorized to run the given function f, and if not it rejects the request. Otherwise, it produces a secret key for the function f as follows. The function f is wrapped in a *function enclave* program, described in more details below. The $\mathsf{Authority}$ generates an instance of this function enclave and obtains an attestation report for the enclave including the MRENCLAVE value $\mathsf{mrenclave}_f$. It then uses the KME signing handle to sign $\mathsf{mrenclave}_f$ using $\mathsf{sk}_{\mathsf{sign}}$. The signature sig_f is returned to the client application, and serves as the "secret key" sk_f.

FE.Encrypt Inputs are encrypted with $\mathsf{pk}_{\mathsf{pke}}$ using a CCA2 secure public key encryption scheme.

FE.Decrypt Decryption begins with a client application connecting to a decryption node that has already been provisioned with the decryption key $\mathsf{sk}_{\mathsf{pke}}$. The client application may also run locally on the decryption node. The following steps ensue:

1. If this is the client's first request to decrypt the function f, the client sends the function enclave binary $\mathsf{enclave}_f$ to the decryption node, which the decryption node then runs. Note that the binary $\mathsf{enclave}_f$ is initialized by untrusted code running on the decryption node, not by the DE.

2. The client initiates a key exchange with the function enclave, and receives a remote attestation that it has successfully established a secure channel with an Intel SGX enclave running $\mathsf{enclave}_f$. (Local client applications skip this step).

3. The client sends over the established secure channel a vector of ciphertexts and the KME signature sig_f that it obtained from the $\mathsf{Authority}$ in $\mathsf{FE.Keygen}$.

4. The function enclave locally attests to the DE and passes sig_f. The DE validates this signature against $\mathsf{vk}_{\mathsf{sign}}$ and the MRENCLAVE value $\mathsf{mrenclave}_f$, which it obtains during local attestation. If this validation passes, the DE delivers the secret key $\mathsf{sk}_{\mathsf{pke}}$ to the function enclave. The DE authenticates its message to the function enclave by wrapping it inside its own local attestation report.[2] Finally, the function enclave uses $\mathsf{sk}_{\mathsf{pke}}$ to decrypt the ciphertexts and compute f on the plaintext values. The output is returned to the client application over the function enclave's secure channel with the client application.

[2] Authenticating the DE's message to the function enclave serves the same purpose as authenticating the KME's message to the DE in the formal proof of security.

4.5 Implementation and evaluation

We implemented a prototype of the Iron system with a single decryption node and a client application running locally on the decryption node. The implementation was developed in C++ using the Intel(R) SGX SDK 1.6 for Windows[3]. All enclaves link the MSR Elliptic Curve Cryptography Library 2.0 MSR_ECClib.lib[4] as a trusted static library, which is used to implement the elliptic curve ElGamal cryptosystem in a Weierstrass curve over a 256-bit prime field, and sgx_tcrypto.lib, which includes EC256-DHKE key exchange, ECDSA signatures over the NIST P-256 elliptic curve, Rijndael AES-GCM encryption on 128-bit key sizes, and SHA256. We implemented a CCA2-secure hybrid encryption scheme using ElGamal, AES-GCM, and SHA256 in the standard way. We tested the prototype implementation on a platform running an Intel Skyake i7-6700 processor at 3.40 GHz with 8 GiB of RAM and Windows Server 2012 R2 Standard operating system, compiled with 64-bit and Debug mode build configurations.

We evaluate three special cases of functional encryption: *identity based encryption* (IBE), *order revealing encryption* (ORE), and *three input DNF* (3DNF). We chose these primarily to demonstrate how our SGX assisted versions of these primitives perform in comparison to purely cryptographic versions that have been implemented, ranging from a widely-used and practical construction (IBE from pairings) to impractical ones (ORE and 3DNF from multilinear maps). Our evaluation confirms that the SGX-based functional encryption examples we implemented are orders of magnitude faster than cryptographic solutions without trusted hardware, even for IBE which is already widely used in practice. We recognize that more complex functionalities than the ones we have implemented, particularly functions that operate on data outside the EPC, may require additional side-channel mitigation techniques such as ORAM, which will impact performance. However, we would still expect these to outperform traditional functional encryption by orders of magnitude.

Side-channel resilience The function and decryption enclave programs must be implemented to resist the software based side-channel attacks on SGX described in Section 3.1.5. The only enclave operations that touch secret data are decryption operations (AES-GCM and ElGamal) and the specific client functions that are loaded into the function enclave. Our implementation of AES-GCM uses the SGX SDK cryptographic library,

[3]https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf

[4]https://www.microsoft.com/en-us/research/project/msr-elliptic-curve-cryptography-library

which calls the AES-NI instruction for AES-GCM, and hence is resilient to software-based side-channels. Our implementation of ElGamal decryption uses the MSR Elliptic Curve Cryptography Library 2.0, which also claims resistance to timing attacks and cache-timing attacks. We implemented data-oblivious versions of all three client-loaded functions that we include in our evaluation, hence these functions also do not leak information about the secret data. This was easy to achieve by implementing data comparisons in x86 assembly with the `setg` and `sete` conditional instructions (similar to [173]).

4.5.1 Implemented ECALLS

Systems using SGX have to be programmed following a specific framework. Enclaves contain trusted function calls (ECALLs) that are executed in enclaves and called from the untrusted application. Untrusted function calls (OCALLs) are defined by the application and may be called from within an enclave. Our FE protocol is implemented using the following set of ECALLs and OCALLs.

KeyManager.dll ECALLs

- `ecdsa_setup` generates a public and private key for 256-bit ECDSA and seals the private key using the SDK function `sgx_seal_data` (this retrieves the enclave's *seal key* via the EGETKEY instruction and AES encrypts the data). The public verification key `vk` and sealed signing key `sk` are returned to the application and written to a file. The ECDSA key generation `sgx_ecc256_create_key_pair` is implemented in sgx_tcrypto.lib.

- `elgamal_setup` generates an ElGamal public key `pubkey` and private key `privkey`. It signs the ElGamal public key with the TA's ECDSA private key and seals the ElGamal private key with the SDK function `sgx_seal_data` (this wraps the EGETKEY instruction to retrieve the Seal Key and AES-GCM encrypts the data). The signed public key and sealed private key are returned to the application and written to a file.

- `sign_function` takes an input array of data (a 256 bit measurement MRENCLAVE) and outputs a 256-bit ECDSA signature on this input using the key `sk` generated in `ecdsa_setup`.

- km_ra_proc receives as input a EC256-DHKE key share ga, an enclave quote structure de_quote. It checks that the report inside de_quote has the appropriate HW configuration, that its mr_enclave field matches the expected MRENCLAVE value of the DE, and that it also includes a 512 byte field report_data containing the value ga. The quote structure also includes an EPID signature on the report, which must be verified with the Intel Attestation Service (see [135]). (This additional procedure is not implemented in our prototype). If all the checks pass, the function generates a EC256-DHKE key share gb, computes the shared EC256-DHKE key, derives from it a 128-bit session key ss_key, and encrypts both the ElGamal private key privkey and the ECDSA verification key vk under ss_key with Rijndael AES-GCM. Finally, it returns gb and the encrypted secret.

FEDecryption.dll ECALLs

- sgx_ra_get_msg first calls the SDK function sgx_ecc256_create_key_pair to generate an EC256-DHKE key share ga. Next it calls the SDK function sgx_init_quote (this contacts the Intel Provisioning Server if the processor has not yet been provisioned with an EPID key). It calls the SDK function sgx_get_quote to obtain the quote structure de_quote (through a local attestation with the Quoting Enclave). It outputs ga and de_quote.

- proc_ra_response receives as input a EC256-DHKE key share gb and an encrypted ElGamal private key privkey. It computes the shared EC256-DHKE key with the SDK function sgx_ecc256_compute_shared_dhkey, derives the 128-bit session key ss_key, and uses it to decrypt privkey with Rijndael AES-GCM. Finally, it seals the decrypted key with sgx_seal_data and outputs the sealed key.

- proc_local_attest receives inputs a EC256-DHKE key share ga, a CMACed enclave report fe_report, and an ECDSA signature fe_report_signature. It verifies the CMAC on fe_report with sgx_verify_report (an SDK function that wraps the EGETKEY instruction to retrieve the Report Key and computes the CMAC). It then verifies (with the KME's verification key) that fe_report_signature is a valid signature on the mr_enclave field of fe_report. If these verifications pass, it generates a EC256-DHKE key share gb, computes the shared EC256-DHKE key, derives a 128-bit shared key aek and encrypts the ElGamal private key privkey under aek with Rijndael AES-GCM. It returns gb and the encrypted privkey.

FEFunction.dll ECALLs

- `local_attest_to_decryption_enclave` generates a EC256-DHKE key share `ga` and calls `sgx_create_report` (an SDK function that wraps the EREPORT instruction) to generate `fe_report`, a CMACed enclave report. The values `ga`, `fe_report`, and `fe_report_signature` are passed to the OCALL `request_local_dh_session_ocall`. It receives back a EC256-DHKE key share `gb` and encrypted ElGamal private key `privkey`. It computes the shared EC256-DHKE key, derives a 128-bit shared key `aek` and decrypts `privkey`. The decrypted key is stored in a static variable.

- `decrypt_order` takes as input a pair of ciphertexts (encrypted integers) and returns 1 if the first integer is less than the second, otherwise 0.

- `decrypt_ibe` In IBE, plaintexts consist of tagged payloads, i.e. have the form (tag, m), and decrypting a ciphertext requires a key specific to the value of tag (i.e. to the corresponding function F_{tag}). To avoid creating a separate enclave for each key issued by the Authority, we have a single ECALL `decrypt_ibe` that multiplexes over all possible tags. This takes as input a ciphertext and a signature sig_{tag}. The Authority will issue sig_{tag} as part of the key for F_{tag} (this is in addition to the signature on the MRENCLAVE value of FEFunction.dll). The ECALL `decrypt_ibe` decrypts the ciphertext to obtain (tag, m), uses the public verification key to check that sig_{tag} is a valid signature on tag, and if so outputs m (otherwise it returns an error).

- `decrypt_3dnf` takes three ciphertexts as input, which are encryptions of n-bit inputs $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, and $z = z_1 \cdots z_n$. It outputs $(x_1 \wedge y_1 \wedge z_1) \vee \cdots \vee (x_n \wedge y_n \wedge z_n)$.

4.5.2 Performance evaluation

We report on the performance of FE.Decrypt, FE.Setup, and FE.Keygen (Figures 4.2 and 4.3). FE.Encrypt in our system is standard public key encryption (our implementation uses ElGamal), and this is done outside of SGX enclaves.[5]

Figure 4.2 contains a break down of the run time for FE.Setup and FE.Keygen.

[5] Note that all the procedures we evaluate are entirely local, which is why we do not include any network performance metrics. We omit performance measures on decryption node setup since the setup procedure requires contacting the Intel Attestation Server to process a remote attestation, which we were unable to test without a license from Intel. Nonetheless, the setup is a one-time operation that is completed when a decryption node platform is first established, and thus has little overall impact on decryption performance.

create enclave	57 ms
ECDSA setup	74 ms
ElGamal setup	8 ms
server setup	2 ms
sign message	11 ms
Total	141 ms

Figure 4.2: FE.Setup and FE.Keygen run time.
FE.Setup includes enclave creation and generation of public/secret keys for ECDSA and ElGamal on 256 bit EC curves. FE.Keygen corresponds to sign message, which generates an ECDSA signature on a 256-bit input.

We evaluated the performance of FE.Decrypt for three special cases of function encryption: *identity based encryption* (IBE), *order revealing encryption* (ORE), and *three input DNF* (3DNF). We chose these functionalities primarily to demonstrate how our SGX assisted versions of these primitives perform in comparison to their purely cryptographic versions (IBE from pairings, DNF and 3DNF from multilinear maps). The table in Figure 4.3 summarizes the decryption times for the three functionalities, including a breakdown of the time spent on the three main ECALLS of the decryption process: enclave creation, local attesting to the DE, and finally decrypting the ciphertext and evaluating the function.

Functionality:	**IBE**	**ORE**	**3DNF**
create enclave	14.5 ms	20.7 ms	19.7 ms
local attest	1.6 ms	2.1 ms	2.1 ms
decrypt & eval	0.98 ms	0.84 ms	0.96 ms
Total	17.8 ms	23.78 ms	22.76 ms

Figure 4.3: Breakdown of FE.Decrypt run times for each of our SGX-FE implementations of IBE, ORE, and 3DNF.
The input in IBE consisted of a 3-byte tag and a 32-bit integer payload. The input pairs in ORE were 32-bit integers, and the input triplets in 3DNF were 16-bit binary strings. (The input types were chosen for consistency with the 5Gen experiments). The column decrypt & eval gives the cost of running a single decryption.

Amortized decryption costs As shown in Figure 4.3, for each of the functionalities the time spent creating the enclave dominates the time spent on decryption and evaluation by 2 orders of magnitude. Once the function enclave has been created and local attestation to the DE is complete, the same enclave can be used to decrypt an arbitrary number of input

	IBE^{SGX}	**IBE**$^{[BF01]}$	$\times$ increase
|msg|	35 bits	35 bits	NA
|c|	175 bytes	471 bytes	2.69
decrypt	17.8 ms	49 ms	2.75
decrypt*	0.39 ms	49 ms	125.64

	ORE^{SGX}	**ORE**^{5Gen}	$\times$ increase
|msg|	32 bits	32 bits	NA
|c|	172 bytes	4.7 GB	$27.3 \cdot 10^6$
decrypt	23.78 ms	4 m	$10.1 \cdot 10^3$
decrypt*	0.32 ms	4 m	$750 \cdot 10^3$

	3DNF^{SGX}	**3DNF**^{5Gen}	$\times$ increase
|msg|	16 bits	16 bits	NA
|c|	170 bytes	2.5 GB	$14.7 \cdot 10^6$
decrypt	22.76 ms	3 m	$7.9 \cdot 10^3$
decrypt*	0.45 ms	3 m	$400 \cdot 10^3$

Figure 4.4: Comparison of decryption times and ciphertext sizes for the SGX-FE implementation of IBE, ORE, 3DNF to cryptographic implementations.
The 5Gen ORE and 3DNF implementation referenced here uses the CLT mmap with an 80-bit security parameter. The column decrypt gives the cost of running a single decryption, and decrypt* gives the amortized cost (per ciphertext tuple) of 10^3 decryptions.

ciphertext tuples. Thus, the amortized cost of running decryption on many ciphertexts (or tuples of ciphertexts) is much lower than the cost of running decryption on a single input. (This is not the case with cryptographic implementations of these functionalities). The amortized cost of running decryption on 1000 inputs (ciphertext tuples) is included in the next table, Figure 4.4.

Comparison to cryptographic implementations We measured decryption time for an implementation[6] of Boneh-Franklin IBE [41] on our platform. We also include decryption time performance numbers for the 5Gen implementation[7] of mmap-based ORE and 3DNF as reported in [147]. We did not deem it necessary to measure 5Gen implementations of ORE and 3DNF on our platform since their performance is 4 orders

[6]The Stanford IBE command-line utility ibe-0.7.2-win, available at https://crypto.stanford.edu/ibe/download.html

[7]5Gen, available https://github.com/5GenCrypto

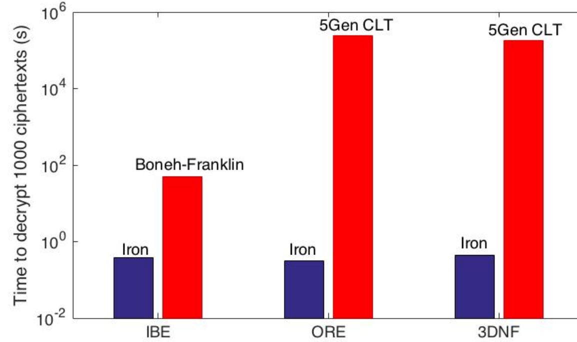


Figure 4.5: Comparison of time for decrypting 10^3 ciphertext tuples using the SGX-FE implementation of IBE, ORE, 3DNF vs cryptographic implementations from pairings and mmaps respectively.

of magnitude slower than that of our SGX-based implementation. The comparison for these multi-input functionalities simply illustrates how our SGX-FE system makes possible primitives that are currently otherwise infeasible to build for practical use without trusted hardware.

4.6 Formalization of Iron

4.6.1 Formal definition of Functional Encryption

We adapt the definition (Definition 2.7.1) of functional encryption to fit the computational model of our system. Interaction with local enclaves is modeled as calls to the HW functionality defined in Definition 3.2.1. Communication with the remote KME is modeled with a separate oracle $\mathsf{KM}(\cdot)$. We allow for a preprocessing phase which runs the setup for all HW instances. A functional encryption scheme $\mathcal{FE}$ for a family of programs $\mathcal{P}$ and message space $\mathcal{M}$ consists of algorithms $\mathcal{FE} = (\mathsf{FE.Setup}, \mathsf{FE.Keygen}, \mathsf{FE.Enc}, \mathsf{FE.DecSetup}, \mathsf{FE.Dec})$ defined as follows.

- $\mathsf{FE.Setup}(1^\lambda)$: On input security parameter λ (in unary), output the master public key mpk and the master secret key msk.
- $\mathsf{FE.Keygen}(\mathsf{msk}, P)$: On input the master secret key msk and a program $P \in \mathcal{P}$, output the secret key sk_P for P.

- $\mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$: On input the master public key mpk and an input message $\mathsf{msg} \in \mathcal{M}$, output a ciphertext ct.
- $\mathsf{FE.DecSetup}^{\mathsf{KM}(\cdot),\mathsf{HW}(\cdot)}(\mathsf{mpk})$: The decryption node setup algorithm has access to the KM oracle and the HW oracles. On input the master public key mpk, output a handle hdl to be used by the actual decryption algorithm.
- $\mathsf{FE.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{hdl}, \mathsf{sk}_P, \mathsf{ct})$: On input a handle hdl for an enclave, a secret key sk_P and a ciphertext ct and outputs $P(\mathsf{msg})$ or $\perp$. This algorithm has access to the interface for all the algorithms of the trusted hardware HW.

Correctness A functional encryption scheme $\mathcal{FE}$ is correct if for all $P \in \mathcal{P}$ and all $\mathsf{msg} \in \mathcal{M}$, the probability for $\mathsf{FE.Dec}^{\mathsf{HW}(\cdot)}\big(\mathsf{hdl}, \mathsf{sk}_P, \mathsf{ct}\big)$ to be not equal to $P(\mathsf{msg})$ is $\mathrm{negl}(\lambda)$, where $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$, $\mathsf{sk}_P \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, P)$, $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$ and $\mathsf{hdl} \leftarrow \mathsf{FE.DecSetup}^{\mathsf{KM}(\cdot),\mathsf{HW}(\cdot)}(\mathsf{mpk})$ and the probability is taken over the random coins of the probabilistic algorithms $\mathsf{FE.Setup}, \mathsf{FE.Keygen}, \mathsf{FE.Enc}, \mathsf{FE.DecSetup}$.

Non-interaction Non-interaction is central to the standard notion of functional encryption. Our construction of hardware assisted FE requires a one-time setup operation where the decryptor's hardware contacts the KME to receive a secret key. However, this interaction only occurs once in the setup of a decryption node, and thereafter decryption is non-interactive. To capture this restriction on interaction we add to the standard FE algorithms an additional algorithm $\mathsf{FE.DecSetup}$, which is given oracle access to a Key Manager $\mathsf{KM}(\cdot)$. The decryption algorithm $\mathsf{FE.Dec}$ is only given access to HW.

Security definition Here, we define a strong simulation-based security of FE similar to [37, 106, 7]. In this security model, a polynomial time adversary will try to distinguish between the real world and a "simulated" world. In the real world, algorithms work as defined in the construction. In the simulated world, we will have to construct a polynomial time simulator which has to do the experiment given only the program queries P made by the adversary and the corresponding results $P(\mathsf{msg})$.

Definition 4.6.1 (SimSecurity-FE)**.** *Consider a stateful simulator $\mathcal{S}$ and a stateful adversary $\mathcal{A}$. Let $U_{\mathsf{msg}}(\cdot)$ denote a universal oracle, such that $U_{\mathsf{msg}}(P) = P(\mathsf{msg})$.*

Both games begin with a pre-processing phase executed by the environment. In the ideal game, pre-processing is simulated by $\mathcal{S}$. Now, consider the following experiments.

$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda)$:

1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot)}(\mathsf{mpk})$
3. $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$
4. $\alpha \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot),\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$
5. *Output* (msg, α)

$\mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$:

1. $\mathsf{mpk} \leftarrow \mathcal{S}(1^\lambda)$
2. $\mathsf{msg} \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(\mathsf{mpk})$
3. $\mathsf{ct} \leftarrow \mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(1^\lambda, 1^{|\mathsf{msg}|})$
4. $\alpha \leftarrow \mathcal{A}^{\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$
5. *Output* (msg, α)

In the above experiment, oracle calls by $\mathcal{A}$ to the key-generation, HW *and* KM *oracles are all simulated by the simulator $\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)$. An FE scheme is simulation-secure against adaptive adversaries if there is a stateful probabilistic polynomial time simulator $\mathcal{S}$ that on each* FE.Keygen *query P queries its oracle $U_{\mathsf{msg}}(\cdot)$ only on the same P (and hence learn just $P(\mathsf{msg})$), such that for every probabilistic polynomial time adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.*

$$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda) \stackrel{c}{\approx} \mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$$

Note that the above definition handles one message only. This can be extended to a definition of security for many messages by allowing the adversary to adaptively output many messages while providing him the ciphertext for a message whenever he outputs one. Here, the simulator will have an oracle $U_{\mathsf{msg}_i}(\cdot)$ for every msg_i output by the adversary.

Simulating HW As previously discussed, we let the simulator intercept all the adversary's queries to HW and return simulated responses, just as in [72]. If we do not allow simulation of HW, it is impossible to achieve Definition 4.6.1. The modified FE definition that we provide in Definition 4.6.2 allows "minimal" interaction[8] with an efficient KM oracle during every run of FE.Dec. In Section 4.8, we give a second construction that realizes this modified FE definition.

Definition 4.6.2 (StrongSimSecurity-FE)**.** *Consider a stateful simulator $\mathcal{S}$ and a stateful adversary $\mathcal{A}$. Let $U_{\mathsf{msg}}(\cdot)$ denote a universal oracle, such that $U_{\mathsf{msg}}(P) = P(\mathsf{msg})$.*

Both games begin with a pre-processing phase executed by the environment. In the ideal game, pre-processing is simulated by $\mathcal{S}$. Now, consider the following experiments.

[8]Allowing unbounded interaction would lead to trivial constructions where KM simply decrypts the ciphertext and returns the function of the message.

$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda)$:

1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot)}(\mathsf{mpk})$
3. $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$
4. $\alpha \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot),\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$
5. *Output* (msg, α)

$\mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$:

1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathcal{S}(\mathsf{msk},\cdot)}(\mathsf{mpk})$
3. $\mathsf{ct} \leftarrow \mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(1^\lambda, 1^{|\mathsf{msg}|})$
4. $\alpha \leftarrow \mathcal{A}^{\mathsf{HW},\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$
5. *Output* (msg, α)

In the above experiment, oracle calls by $\mathcal{A}$ to the key-generation and KM *oracles are simulated by the simulator $\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)$. But the simulator does not simulate the* HW *algorithms, except* HW.Setup*. We call a simulator* admissible *if on each input P, it just queries its oracle $U_{\mathsf{msg}}(\cdot)$ on P (and hence learns just $P(\mathsf{msg})$).*

The FE scheme is said to be simulation-secure against adaptive adversaries if there is an admissible *stateful probabilistic polynomial time simulator $\mathcal{S}$ such that for every probabilistic polynomial time adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.*

$$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda) \overset{c}{\approx} \mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$$

4.6.2 Crypto primitive definitions

We recall the formal definitions of some fundamental cryptographic primitives

Secret key encryption A secret key encryption scheme E supporting a message domain $\mathcal{M}$ consists of a probabilistic polynomial time key generation algorithm $\mathsf{E.KeyGen}(1^\lambda)$ that takes in a security parameter and outputs a key sk from the key space $\mathcal{K}$, a probabilistic polynomial time encryption algorithm $\mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg})$ that takes in a key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the ciphertext ct, and a deterministic polynomial time decryption algorithm $\mathsf{E.Dec}(\mathsf{sk}, \mathsf{ct})$ that takes in a key sk and a ciphertext ct and outputs the decryption msg.

A secret key encryption scheme E is correct if for all λ and all $\mathsf{msg} \in \mathcal{M}$,

$$\Pr\left[\mathsf{E.Dec}\Big(\mathsf{sk}, \mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg})\Big) \neq \mathsf{msg} \Big| \mathsf{sk} \leftarrow \mathsf{E.KeyGen}(1^\lambda)\right] = \mathrm{negl}(\lambda)$$

where the probability is taken over the random coins of the probabilistic algorithms $\mathsf{E.KeyGen}, \mathsf{E.Enc}$.

A secret key encryption scheme E is said to have *indistinguishability security under chosen plaintext attack* (IND-CPA) if there is no polynomial time adversary $\mathcal{A}$ which can win the following game with probability non-negligible in λ:

Definition 4.6.3. (IND-CPA security of E). *We define the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *The challenger run the* $\mathsf{E.KeyGen}$ *algorithm to obtain a key* sk *from the key space* $\mathcal{K}$.
2. *The challenger also chooses a random bit* $b \in \{0, 1\}$.
3. *Whenever the adversary provides a pair of messages* $(\mathsf{msg}_0, \mathsf{msg}_1) \in \mathcal{M}^2$ *of its choice, the challenger replies with* $\mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg}_b)$.
4. *The adversary finally outputs its guess* b'.

The advantage of adversary in the above game is

$$\mathsf{Adv}_{\mathsf{Enc}}(\mathcal{A}) := \Pr[b' = b] - \frac{1}{2}$$

A signature scheme A digital signature scheme S supporting a message domain $\mathcal{M}$ consists of a probabilistic polynomial time algorithm $\mathsf{S.KeyGen}(1^\lambda)$ that takes in a security parameter and outputs the signing key sk and a verification key vk, a probabilistic polynomial time signing algorithm $\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$ that takes in a signing key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the signature σ, and a deterministic verification algorithm $\mathsf{S.Verify}(\mathsf{vk}, \sigma, \mathsf{msg})$ that takes in a verification key vk, a signature σ and a message msg and outputs 0 or 1.

A signature scheme S is correct if for all $\mathsf{msg} \in \mathcal{M}$,

$$\Pr\left[\mathsf{S.Verify}\Big(\mathsf{vk}, \mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg}), \mathsf{msg}\Big) = 0 \Big| (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)\right] = \mathrm{negl}(\lambda)$$

where the probability is taken over the random coins of the probabilistic algorithms $\mathsf{S.KeyGen}, \mathsf{S.Sign}$.

A signature scheme S is said to be *existentially unforgeable under chosen message attack* (EUF-CMA) if there is no polynomial time adversary which can win the following game with probability non-negligible in λ.

Definition 4.6.4. (EUF-CMA security of $\mathcal{S}$). *We define the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *The challenger runs the* $\mathsf{S.KeyGen}$ *algorithm to obtain the key pair* $(\mathsf{sk}, \mathsf{vk})$*, and provides the verification key* vk *to the adversary.*
2. *Initialize* $\mathsf{query} = \{\}$.
3. *Now, whenever the adversary provides a query with a message* msg*, the challenger replies with* $\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$. *Also,* $\mathsf{query} = \mathsf{query} \cup \{\mathsf{msg}\}$.
4. *Finally, the adversary outputs a forged signature* σ^* *corresponding to a message* msg^*.

The advantage of $\mathcal{A}$ *in the above security game is*

$$\mathsf{Adv}_{\mathsf{sign}}(\mathcal{A}) := \Pr\left[\mathsf{S.Verify}(\mathsf{vk}, \sigma^*, \mathsf{msg}^*) = 1 \middle| \mathsf{msg}^* \notin \mathsf{query}\right]$$

Public key encryption A public key encryption (PKE) scheme supporting a message domain $\mathcal{M}$ consists of a probabilistic polynomial time algorithm $\mathsf{PKE.KeyGen}(1^\lambda)$ that takes in a security parameter and outputs a key pair $(\mathsf{pk}, \mathsf{sk})$, a probabilistic encryption algorithm $\mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg})$ that takes in a public key pk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs a ciphertext ct, and a deterministic decryption algorithm $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$ that takes in a secret key sk and a ciphertext ct and outputs the decryption msg or $\perp$.

A PKE scheme PKE is correct if for all λ and $\mathsf{msg} \in \mathcal{M}$,

$$\Pr\left[\mathsf{PKE.Dec}\Big(\mathsf{sk}, \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg})\Big) \neq \mathsf{msg} \middle| (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)\right] = \mathrm{negl}(\lambda)$$

where the probability is taken over the random coins of the probabilistic algorithms $\mathsf{KeyGen}, \mathsf{Enc}$.

A PKE scheme provides confidentiality to the encrypted message. Formally, a PKE scheme PKE is said to have *indistinguishability security under adaptively chosen ciphertext attack* (IND-CCA2) if there is no polynomial time adversary $\mathcal{A}$ which can guess $b' = b$ in the following game with probability non-negligible in λ, *plus* half.

Definition 4.6.5. (IND-CCA2 security of PKE). *We define the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. $\mathcal{C}$ *runs the* $\mathsf{PKE.KeyGen}$ *algorithm to obtain a key pair* $(\mathsf{pk}, \mathsf{sk})$ *and gives* pk *to the adversary.*

2. $\mathcal{A}$ *provides adaptively chosen* ct *and gets back* $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$*.*

3. $\mathcal{A}$ *provides* $\mathsf{msg}_0, \mathsf{msg}_1$ *to* $\mathcal{C}$*.*

4. $\mathcal{C}$ *then runs* $\mathsf{PKE.Enc}(\mathsf{pk})$ *to obtain* $\mathsf{ct}^* = \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg}_b)$ *for* $b \xleftarrow{\$} \{0,1\}$*.* $\mathcal{C}$ *provides* ct^* *to* $\mathcal{A}$*.*

5. $\mathcal{A}$ *continues to provide adaptively chosen* ct *multiple times and gets back* $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$*, with a restriction that* $\mathsf{ct} \neq \mathsf{ct}^*$*.*

6. $\mathcal{A}$ *outputs its guess* b'*.*

The advantage of the adversary $\mathcal{A}$ *in the above game is*

$$\mathsf{Adv}_{\mathsf{pke}}(\mathcal{A}) := \Pr[b' = b] - \frac{1}{2}$$

A PKE scheme may also be "weakly robust" [2]. Informally, this means that a ciphertext when decrypted with an "incorrect" secret key should output $\perp$ when all the algorithms are honestly run.

Definition 4.6.6. ((Weak) robustness property of PKE). *A PKE scheme* PKE *has the (weak) robustness property if for all* λ *and* $\mathsf{msg} \in \mathcal{M}$*,*

$$\Pr\Big[\mathsf{PKE.Dec}\Big(\mathsf{sk}', \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg})\Big) \neq \perp\Big] = \mathrm{negl}(\lambda)$$

where $(\mathsf{pk}, \mathsf{sk})$ *and* $(\mathsf{pk}', \mathsf{sk}')$ *are generated by running* $\mathsf{PKE.KeyGen}(1^\lambda)$ *twice, and the probability is taken over the random coins of the probabilistic algorithms* $\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}$*.*

We will use this property in our construction in the stronger security model.

4.6.3 FE Formal construction

We present here the formal description of our FE system using the syntax of the HW model from Definition 3.2.1. The trusted authority platform TA and decryption node platform DN each have access to instances of HW. Let PKE denote an IND-CCA2 secure public key encryption scheme (Definition 4.6.5) and let S denote an existentially unforgeable signature scheme (Definition 4.6.4).

Pre-processing phase TA and DN run $\mathsf{HW.Setup}(1^\lambda)$ for their HW instances and record the output params.

$\mathbf{FE.Setup}^{\mathsf{HW}}(1^\lambda)$ The key manager enclave program Q_{KME} is defined in Figure 4.6. The value tag_{DE}, the measurement of the program Q_{DE}, is hardcoded in the static data of Q_{KME}. Let state denote an internal state variable.

Figure 4.6: Q_{KME}

- On input ("init", 1^λ):
  1. Run $(\mathsf{pk_{pke}}, \mathsf{sk_{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ and $(\mathsf{vk_{sign}}, \mathsf{sk_{sign}}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)$
  2. Update state to $(\mathsf{sk_{pke}}, \mathsf{sk_{sign}}, \mathsf{vk_{sign}})$ and output $(\mathsf{pk_{pke}}, \mathsf{vk_{sign}})$
- On input ("provision", $\mathsf{quote}, \mathsf{params}$):
  1. Parse $\mathsf{quote} = \big(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \sigma\big)$, check that $\mathsf{tag}_Q = \mathsf{tag}_{DE}$. If not, output $\perp$.
  2. Parse $\mathsf{input} =$ ("init setup", $\mathsf{vk_{sign}}$) and check if $\mathsf{vk_{sign}}$ matches with the one in state. If not, output $\perp$.
  3. Parse $\mathsf{output} = (\mathsf{sid}, \mathsf{pk})$ and run $b \leftarrow \mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$ on quote. If $b = 0$ output $\perp$.
  4. Retrieve $\mathsf{sk_{pke}}$ from state and compute $\mathsf{ct}_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{sk_{pke}})$ and $\sigma_{sk} = \mathsf{S.Sign}(\mathsf{sk_{sign}}, (\mathsf{sid}, \mathsf{ct}_{sk}))$ and output $(\mathsf{sid}, \mathsf{ct}_{sk}, \sigma_{sk})$.
- On input ("sign", msg):
  Compute $\mathsf{sig} \leftarrow \mathsf{S.Sign}(\mathsf{sk_{sign}}, \mathsf{msg})$ and output sig.

1. Run $\mathsf{hdl}_{KME} \leftarrow \mathsf{HW.Load}(\mathsf{params}, Q_{KME})$.
2. Run $(\mathsf{pk_{pke}}, \mathsf{vk_{sign}}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"init"}, 1^\lambda))$.
3. Output the master public key $\mathsf{mpk} := (\mathsf{pk_{pke}}, \mathsf{vk_{sign}})$ and the master secret key $\mathsf{msk} := \mathsf{hdl}_{KME}$.

$\textsf{\textbf{FE.Keygen}}^{\textsf{HW}}(\textsf{msk}, P)$

1. Parse $\textsf{msk} = \textsf{hdl}_{KME}$ as a handle to $\textsf{HW.Run}$.
2. Derive $\textsf{tag}_P$ and call $\textsf{sig} \leftarrow \textsf{HW.Run}(\textsf{hdl}_{KME}, (\text{“sign”}, \textsf{tag}_P))$.
3. Output $\textsf{sk}_p := \textsf{sig}$.

$\textsf{\textbf{FE.Enc}}(\textsf{mpk}, \textsf{msg})$

1. Parse $\textsf{mpk} = (\textsf{pk}, \textsf{vk})$.
2. Compute $\textsf{ct} \leftarrow \textsf{PKE.Enc}(\textsf{pk}, \textsf{msg})$.
3. Output $\textsf{ct}$.

$\textsf{\textbf{FE.DecSetup}}^{\textsf{HW},\textsf{KM}(\cdot)}(\textsf{sk}_P, \textsf{ct})$ The decryption enclave program Q_{DE} is defined in Figure 4.7. The security parameter λ is hardcoded into the program.

1. Run $\textsf{hdl}_{DE} \leftarrow \textsf{HW.Load}(\textsf{params}, Q_{DE})$.
2. Parse $\textsf{mpk} = (\textsf{sk}_{\textsf{pke}}, \textsf{vk}_{\textsf{sign}})$ and call $\textsf{quote} \leftarrow \textsf{HW.Run\&Quote}_{\textsf{sk}_{\textsf{HW}}}(\textsf{hdl}_{DE}, \text{“init setup”}, \textsf{vk}_{\textsf{sign}})$.
3. Query $\textsf{KM}(\textsf{quote})$, which internally runs
   $(\textsf{sid}, \textsf{ct}_{sk}, \sigma_{sk}) \leftarrow \textsf{HW.Run}(\textsf{hdl}_{KME}, (\text{“provision”}, \textsf{quote}, \textsf{params}))$.[9]
4. Call $\textsf{HW.Run}(\textsf{hdl}_{DE}, (\text{“complete setup”}, \textsf{sid}, \textsf{ct}_{sk}, \sigma_{sk}))$.
5. Output $\textsf{hdl}_{DE}$.

$\textsf{\textbf{FE.Dec}}^{\textsf{HW}(\cdot)}(\textsf{hdl}, \textsf{sk}_P, \textsf{ct})$ Define a function enclave program parameterized by P as in Figure 4.8.

1. Run $\textsf{hdl}_P \leftarrow \textsf{HW.Load}(\textsf{params}, Q_{FE}(P))$.

[9] We could use HW.Run&Quote here instead of explicitly creating the signature σ_k. If we do that, the verification step in DE would involve using the Intel Attestation Service.

Figure 4.7: Q_{DE}

- On input ("init setup", $\mathsf{vk_{sign}}$):
  1. Run $(\mathsf{pk_{ra}}, \mathsf{sk_{ra}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
  2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
  3. Update state to $(\mathsf{sid}, \mathsf{sk_{ra}}, \mathsf{vk_{sign}})$, and output $(\mathsf{sid}, \mathsf{pk_{ra}})$.
- On input ("complete setup", $\mathsf{sid}, \mathsf{ct}_{sk}, \sigma_{sk}$):
  1. Look up the state to obtain the entry $(\mathsf{sid}, \mathsf{sk_{ra}}, \mathsf{vk_{sign}})$. If no entry exists for sid, output $\perp$.
  2. Verify the signature $b \leftarrow \mathsf{S.Verify}(\mathsf{vk_{sign}}, \sigma_{sk}, (\mathsf{sid}, \mathsf{ct}_{sk}))$. If $b = 0$, output $\perp$.
  3. Run $m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{ra}}, \mathsf{ct}_{sk})$ and parse $m = (\mathsf{sk_{pke}})$.
  4. Add the tuple $(\mathsf{sk_{pke}}, \mathsf{vk_{sign}})$ to state[a].
- On input ("provision", $\mathsf{report}, \mathsf{sig}$):
  1. Check to see that the setup has been completed, i.e. that state contains the tuple $(\mathsf{sk_{pke}}, \mathsf{vk_{sign}})$. If not, output $\perp$.
  2. Check to see that the report has been verified, i.e. that state contains the tuple $(1, \mathsf{report})$. If not, output $\perp$.
  3. Parse $\mathsf{report} = \big(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$ and compute $b \leftarrow \mathsf{S.Verify}(\mathsf{vk_{sign}}, \mathsf{sig}, \mathsf{tag}_Q)$. If $b = 0$, output $\perp$.
  4. Parse output as $(\mathsf{sid}, \mathsf{pk})$. If $b = 1$ output $(\mathsf{sid}, \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{sk_{pke}}))$. Else, output $\perp$.

[a] $\mathsf{vk_{sign}}$ is already in state as part of the outputs of the previous "init setup" phase, but it is useful store and use this tuple as result of a successfully completed setup.

2. Call $\mathsf{report} \leftarrow \mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl}_P, \text{"init"})$.
3. Run $\mathsf{HW.ReportVerify}_{\mathsf{sk_{report}}}\ (\mathsf{hdl}_{DE}, \mathsf{report})$ with $\mathsf{hdl}_{DE} = \mathsf{hdl}$.
4. Call $\mathsf{report}_{sk} \leftarrow \mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, (\text{"provision"}, \mathsf{report}, \mathsf{sig}))$ with $\mathsf{sig} = \mathsf{sk}_P$.

Figure 4.8: $Q_{FE}(P)$

- On input (“init”):
  1. Run $(\mathsf{pk_{la}}, \mathsf{sk_{la}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
  2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
  3. Update state to $(\mathsf{sid}, \mathsf{sk_{la}})$, and output $(\mathsf{sid}, \mathsf{pk_{la}})$.
- On input (“run”, $\mathsf{report}_{sk}, \mathsf{ct}_{msg}$):
  1. Check to see that the report has been verified, i.e. that state contains the tuple $(1, \mathsf{report}_{sk})$. If not, output $\perp$.
  2. Parse $\mathsf{report}_{sk} = \big(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$. Parse output as $(\mathsf{sid}, \mathsf{ct}_{key})$.
  3. Look up the state to obtain the entry $(\mathsf{sid}, \mathsf{sk_{la}})$. If no entry exists for sid, output $\perp$.
  4. Compute $\mathsf{sk_{pke}} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{ra}}, \mathsf{ct}_{key})$ and use it to decrypt $x \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{pke}}, \mathsf{ct}_{msg})$.
  5. Run P on x and record the output $\mathsf{output} := P(x)$. Output output.

5. Run $\mathsf{HW.ReportVerify}_{\mathsf{sk_{report}}}(\mathsf{hdl}_P, \mathsf{report}_{sk})$.
6. Call $\mathsf{output} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_P, \text{“run”}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ with $\mathsf{ct}_{msg} = \mathsf{ct}$.
7. Output output.

4.7 Security

We first explain the crux of our security proof here. More details will follow.

We construct a simulator $\mathcal{S}$ which can simulate $\mathsf{FE.Keygen}$, HW, KM oracles and simulate the challenge ciphertext for the challenge message msg^* provided by the adversary $\mathcal{A}$. The only information that $\mathcal{S}$ will get about msg^* other than its length is the access to the U_{msg^*} oracle which reveals $P(\mathsf{msg}^*)$ for the P's queried by $\mathcal{A}$ to $\mathsf{FE.Keygen}$. At a high level, the proof idea is simple: $\mathcal{S}$ encrypts zeros as the the challenge ciphertext ct^* and $\mathsf{FE.Keygen}$ is

simulated honestly. In the ideal experiment, $\mathcal{S}$ intercepts $\mathcal{A}$'s queries to HW and provides simulated responses. It can use its U_{msg^*} oracle to get $P(\mathsf{msg}^*)$ and simply send this back to $\mathcal{A}$ as the simulated HW output. If $\mathcal{A}$ queries HW on any ciphertexts that do not match the challenge ciphertext ct^*, $\mathcal{S}$ can decrypt them honestly since it possesses msk. Since $\mathcal{S}$ has to modify the program descriptions in enclaves, we provide $\mathcal{S}$ access to the HW keys $\mathsf{sk}_{\mathsf{report}}$ and $\mathsf{sk}_{\mathsf{quote}}$ to produce reports and quotes.

Despite the apparent simplicity, the following subtleties make the proof of security more challenging than on first sight:

1. The simple proof sketch does not account for all of $\mathcal{A}$'s interaction with HW between sending ct^* and receiving back $P(\mathsf{msg}^*)$. HW communicates through $\mathcal{A}$ as a proxy. $\mathcal{A}$ might even tamper with these intermediate messages and observe how HW responds. We need to ensure that anything $\mathcal{A}$ observes in the real experiment can be simulated in the ideal experiment.

2. We use IND-CCA2 public key encryption to secure communication between enclaves that is intercepted by $\mathcal{A}$. $\mathcal{S}$ will need to simulate this communication. Proving that $\mathcal{A}$ cannot distinguish this involves a reduction to the IND-CCA2 security game, showing that if $\mathcal{A}$ can distinguish the real and simulated communication then it would break the IND-CCA2 security. The IND-CCA2 adversary will need to simulate the entire FE system for $\mathcal{A}$ without knowledge of the corresponding secret keys for the public keys that the enclaves are using to secure their communication. In particular, it must see if $\mathcal{A}$ tampers with messages in a way that would cause the system to abort). This is what necessitates an extra layer of authentication on the communication between enclaves

3. The final challenge is that the adversary can also load modified programs of its choice into different enclaves and test their behavior with honest or tampered inputs. This aspect in particular makes the security proof challenging because the FE simulator in the ideal world has to identify whether honest attested programs are running inside the enclaves, and produce simulated outputs only for those enclaves. This gets tricky as there are three enclaves each with multiple entry points.

4.7.1 Security proof

Theorem 4.7.1. *If* S *is an EUF-CMA secure signature scheme,* PKE *is an IND-CCA2 secure public key encryption scheme and* HW *is a trusted hardware scheme, then* FE *is a*

secure functional encryption scheme according to Definition 4.6.1.

Proof. We will construct a simulator $\mathcal{S}$ for the FE security game in Definition 4.6.1. $\mathcal{S}$ is given the length $|\mathsf{msg}^*|$ and an oracle access to $U_{\mathsf{msg}^*}(\cdot)$ (such that $U_{\mathsf{msg}^*}(P) = P(\mathsf{msg}^*)$) after the adversary provides its challenge message msg^*. $\mathcal{S}$ can use this U_{msg^*} oracle on the programs queried by the adversary $\mathcal{A}$ to $\mathsf{FE.Keygen}$. $\mathcal{S}$ has to simulate the pre-processing phase and a ciphertext corresponding to the challenge message msg^* along with answering the adversary's queries to the KeyGen, HW and the KM oracles.

Pre-processing phase: $\mathcal{S}$ simulates the pre-processing phase similar to the real world. $\mathcal{S}$ runs $\mathsf{HW.Setup}(1^\lambda)$ and records $(\mathsf{sk}_{\mathsf{quote}}, \mathsf{sk}_{\mathsf{report}})$ generated during the process. $\mathcal{S}$ measures and stores tag_{DE}. $\mathcal{S}$ also creates empty lists $\mathcal{K}, \mathcal{R}, \mathcal{N}, L_{KM}, L_{DE}, L_{DE2}, L_{FE}$ which will be used later.

FE.Keygen*(msk, P) When $\mathcal{A}$ makes a query to the $\mathsf{FE.Keygen}$ oracle, $\mathcal{S}$ responds the same way as in the real world except that $\mathcal{S}$ now stores all the tag_P corresponding to the P's queried in a list $\mathcal{K}$.

FE.Enc*$(\mathsf{mpk}, 1^{|\mathsf{msg}^*|})$ $\mathcal{S}$ outputs $\mathsf{ct}^* \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, 0^{|\mathsf{msg}^*|})$ and stores ct^* in the list $\mathcal{R}$.

HW oracle For $\mathcal{A}$'s queries to the algorithms of the HW oracle, $\mathcal{S}$ runs the corresponding HW algorithms honestly and outputs their results except for the following oracle calls.

- $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, \text{“provision”}, \mathsf{quote}, \mathsf{params})$: When a provision query is made to KME, $\mathcal{S}$ parses $\mathsf{quote} = \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \sigma\big)$ and outputs $\perp$ if $\mathsf{output} \notin L_{DE2}$. Else, it honestly runs the HW algorithm and then replaces ct_{sk} with $\mathsf{PKE.Enc}(\mathsf{pk}, 0^{|\mathsf{sk}_{\mathsf{pke}}|})$. $\mathcal{S}$ also generates and replaces σ_{sk} for the modified ct_{sk}. Finally, $\mathcal{S}$ stores $(\mathsf{sid}, \mathsf{ct}_{sk})$ in L_{KM}.
- $\mathsf{HW.Load}(\mathsf{params}, Q)$: When the load algorithm is run for a Q corresponding to that of a DE, $\mathcal{S}$ runs the load algorithm honestly and outputs hdl_{DE}. In addition, it stores hdl_{DE} in the list Ψ. When the load algorithm is run for a Q of the form $Q_{FE}(P)$, $\mathcal{S}$ adds the output handle hdl_P to the list $\mathcal{K}$ as follows. $\mathcal{S}$ first checks if the tag_P corresponding to this has an entry in $\mathcal{K}$, and if it exists $\mathcal{S}$ appends hdl_P to its handle list. Else, $\mathcal{S}$ adds the tuple $(0, \mathsf{tag}_P, \mathsf{hdl}_P)$ to $\mathcal{K}$.

- $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"init setup"}, \mathsf{vk_{sign}})$: When an init setup query is made to a $\mathsf{hdl}_{DE} \in \Psi$, $\mathcal{S}$ checks if $\mathsf{vk_{sign}}$ matches with the one in mpk. Else, it removes hdl_{DE} from Ψ. Ψ will remain as the list of handles for DEs with the correct $\mathsf{vk_{sign}}$ fed as input. Then, $\mathcal{S}$ runs $\mathsf{HW.Run}$ honestly on the given input and outputs the result. It also adds $(\mathsf{sid}, \mathsf{pk_{ra}})$ to the list L_{DE2}.

- $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"complete setup"}, \mathsf{sid}, \mathsf{ct}_{sk}, \sigma_{sk})$: When a complete setup query is made to a $\mathsf{hdl}_{DE} \in \Psi$, $\mathcal{S}$ outputs $\perp$ if $(\mathsf{sid}, \mathsf{ct}_{sk}) \notin L_{KM}$. Else, it honestly executes $\mathsf{HW.Run}$. Similar changes are made for $\mathsf{HW.Run\&Report}$ and $\mathsf{HW.Run\&Quote}$ on this set of inputs.

- $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"provision"}, \mathsf{report}, \mathsf{sig})$: When a provision query is made to a $\mathsf{hdl}_{DE} \in \Psi$, $\mathcal{S}$ parses $\mathsf{report} = \big(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$ and outputs $\perp$ if $\mathsf{output} \notin L_{FE}$. Else, it honestly executes $\mathsf{HW.Run}$. At the end, $\mathcal{S}$ adds the output $(\mathsf{sid}, \mathsf{ct}_{key})$ to L_{DE}.

- $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{"init"})$: When an init query is made to a $\mathsf{hdl}_P \in \mathcal{K}$ whose tuple in $\mathcal{K}$ has the honest bit set, $\mathcal{S}$ runs $\mathsf{HW.Run\&Report}$ honestly and outputs the result. It also adds $(\mathsf{sid}, \mathsf{pk_{la}})$ to the list L_{FE}.

- $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$: When a run query is made to $\mathsf{hdl}_P \in \mathcal{K}$ whose tuple in $\mathcal{K}$ has the honest bit set, $\mathcal{S}$ first parses $\mathsf{report}_{sk} = \big(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{input}, \mathsf{output}, \mathsf{mac}\big)$ and outputs $\perp$ if $\mathsf{output} \notin L_{DE}$. Else, it runs $\mathsf{HW.Run}$ on the given inputs. If the output is $\perp$, $\mathcal{S}$ outputs $\perp$. Else, it parses output as $(\mathsf{sid}, \mathsf{ct}_{key})$ and retrieves $\mathsf{sk_{pke}}$ from msk. If $\mathsf{ct}_{msg} \notin \mathcal{R}$, $\mathcal{S}$ computes $x \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{pke}}, \mathsf{ct}_{msg})$, runs P on x and outputs $\mathsf{output} := P(x)$. If $\mathsf{ct}_{msg} \in \mathcal{R}$, $\mathcal{S}$ queries its U_{msg^*} oracle on P and outputs the response.

- For the $\mathsf{HW.Run\&Report}$ and $\mathsf{HW.Run\&Quote}$ queries, similar changes are made as in the respective $\mathsf{HW.Run}$s above. But, report and quote are generated for unmodified tag's of the unmodified programs descriptions. (This is to prevent the adversary from being able to distinguish the change in hybrids just by looking at the report or quote.)

KM oracle For $\mathcal{A}$'s queries to the KM oracle with input quote, $\mathcal{S}$ uses the provision queries to $\mathsf{HW.Run}$ for KME with the changes mentioned above.

Now, for this polynomial time simulator $\mathcal{S}$ described above, we will show that for experiments in Definition 4.6.1,

$$(\mathsf{msg}, \alpha)_{\mathsf{real}} \overset{c}{\approx} (\mathsf{msg}, \alpha)_{\mathsf{ideal}} \tag{4.1}$$

We prove this by showing that the view of the adversary $\mathcal{A}$ in the real world is computationally indistinguishable from its view in the ideal world. It can be easily checked

that the algorithms $\mathsf{KeyGen}^*, \mathsf{Enc}^*$ and oracle KM^* simulated by S correspond to the ideal world specifications of Definition 4.6.1 (because the only information that $\mathcal{S}$ obtains about msg^* is through the $U_{\mathsf{msg}^*}(\cdot)$ oracle which it queries on the $\mathsf{FE.Keygen}$ queries made by $\mathcal{A}$). We will prove through a series of hybrids that $\mathcal{A}$ cannot distinguish between the real and the ideal world algorithms and oracles.

Hybrid 0 $\mathsf{Exp}^{\mathsf{real}}_{\mathbf{FE}}(1^\lambda)$ is run.

Hybrid 1 As in **Hybrid** 0, except that $\mathsf{FE.Keygen}^*$ run by $\mathcal{S}$ is used to generate secret keys instead of $\mathsf{FE.Keygen}$. Also, the ct^* returned by $\mathsf{FE.Enc}$ for the encryption of the challenge message msg^* is stored in the list $\mathcal{R}$. Also, when $\mathsf{HW.Load}(\mathsf{params}, Q)$ is run for the Q of a DE, store the output in the list Ψ, and when $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"init setup"}, \mathsf{vk}_{\mathsf{sign}})$ is run with a $\mathsf{vk}_{\mathsf{sign}}$ different from that in mpk, remove hdl_{DE} from Ψ. Also, when $\mathsf{HW.Load}$ is run for a Q of the form $Q_{FE}(P)$, the output handle hdl_P is added to the list $\mathcal{K}$ in the tuple corresponding to tag_P. If tag_P does not have an entry in $\mathcal{K}$, the entire tuple $(0, \mathsf{tag}_P, \mathsf{hdl}_P)$ is added to $\mathcal{K}$.

Here, $\mathsf{FE.Keygen}^*$ and $\mathsf{FE.Keygen}$ are identical. And storing in lists does not affect the view of $\mathcal{A}$. Hence, **Hybrid** 1 is indistinguishable from **Hybrid** 0.

Hybrid 2 As in **Hybrid** 1, except that when the $\mathsf{HW.Run\&Report}$ is queried with $(\mathsf{hdl}_{DE}, (\text{"provision"}, \mathsf{report}, \mathsf{sig}))$ for $\mathsf{hdl}_{DE} \in \Psi$, $\mathcal{S}$ outputs $\perp$ if tag_P that is part of report does not have an entry in $\mathcal{K}$ with the honest bit set.

If sig is not a valid signature of tag_P, then the $\mathsf{S.Verify}$ step during the execution of $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \cdot)$ would make it output $\perp$. Hence, **Hybrid** 2 differs from **Hybrid** 1 only when a valid signature sig for tag_P is part of the "provision" query to $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \cdot)$ with a hdl_{DE} that has the correct $\mathsf{vk}_{\mathsf{sign}}$ in its state and with a P that $\mathcal{A}$ has not queried to $\mathsf{FE.Keygen}^*$. But, if $\mathcal{A}$ does make a query of this kind to $\mathsf{HW.Run\&Report}$ with a valid sig, Lemma 4.7.2 shows that this can be used to break the existential unforgeability of the signature scheme S.

Hybrid 3.0 As in **Hybrid** 2, except that $\mathcal{S}$ maintains a list L_{KM} of all the "provision" query responses from KM i.e., the $(\mathsf{sid}, \mathsf{ct}_{sk})$ tuples. Then, on any call to $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"complete setup"}, \mathsf{sid}, \mathsf{ct}_k, \sigma_k)$ for $\mathsf{hdl}_{DE} \in \Psi$, if $(\mathsf{sid}, \mathsf{ct}_{sk}) \notin L_{KM}$, $\mathcal{S}$ outputs

$\perp$.

The proof at a high level will be similar to the previous one. $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"complete setup"}, \cdot)$ already outputs $\perp$ in **Hybrid** 2 if σ_{sk} is not a valid signature of $(\mathsf{sid}, \mathsf{ct}_{sk})$ or if an entry for the session ID sid is not in state. So, **Hybrid** 3.0 differs from **Hybrid** 2 only when $\mathcal{A}$ can produce a valid signature σ_{sk} on a $(\mathsf{sid}, \mathsf{ct}_{sk})$ pair for a sid which it has seen before in the communication between KM and a DE whose handle is in Ψ. This is proved in Lemma 4.7.3.

Hybrid 3.1 As in **Hybrid** 3.0, except that $\mathcal{S}$ maintains a list L_{DE} of all the "provision" query responses from $\mathsf{hdl}_{DE} \in \Psi$ i.e., the $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{Q_{DE}}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}))$ tuples. And, on call to $\mathsf{HW.Run}(\mathsf{hdl}_P, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ with hdl_P having an entry in $\mathcal{K}$ with its honest bit set, $\mathcal{S}$ outputs $\perp$ if $\mathsf{report}_{sk} = \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{input}, (\mathsf{sid}, \mathsf{ct}_{key}), \mathsf{mac}\big)$ with $\mathsf{tag}_Q = \mathsf{tag}_{DE}$, sid having an entry in state and $(\mathsf{sid}, \mathsf{ct}_{key}) \notin L_{DE}$.

Local attestation helps in proving the indistinguishability of the hybrids. For honest hdl_Ps, $\mathsf{HW.Run}(\mathsf{hdl}_P, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ already outputs $\perp$ in **Hybrid** 3.0 if for $\mathsf{report}_{sk} = \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{DE}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}), \mathsf{mac}\big)$, mac is not a valid MAC on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{DE}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}))$, or if sid does not have an entry in state. So, the only change in **Hybrid** 3.1 is that $\mathsf{HW.Run}$ also outputs $\perp$ if mac is a valid MAC but on a $(\mathsf{sid}, \mathsf{ct}_{key}) \notin L_{DE}$. Hence, $\mathcal{A}$ can distinguish between the hybrids only when it produces a valid mac on a tuple with $(\mathsf{sid}, \mathsf{ct}_{sk})$ not in L_{DE}. But this happens with negligible probability due to the security of local attestation.

Hybrid 4 As in **Hybrid** 3.1, except that when $\mathsf{HW.Run}$ is queried with $(\mathsf{hdl}_P, \text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ where report_{sk} is a valid MAC of a tuple containing an entry in L_{DE} and $\mathsf{hdl}_P \in \mathcal{K}$ with the honest bit set. If $\mathsf{ct}_{msg} \in \mathcal{R}$, $\mathcal{S}$ uses the U_{msg^*} oracle to answer the $\mathsf{HW.Run}$ query. If $\mathsf{ct}_{msg} \notin \mathcal{R}$, $\mathcal{S}$ uses the $\mathsf{sk}_{\mathsf{pke}}$ from $\mathsf{FE.Setup}$ to decrypt ct_{msg} instead of the one got by decrypting ct_{key} i.e.,

- On input $(\text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$:

  4. If $\mathsf{ct}_{msg} \notin \mathcal{R}$, retrieve $\mathsf{sk}_{\mathsf{pke}}$ from msk. Compute $x \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_{msg})$. Run P on x and record the output $\mathsf{output} := P(x)$. Output output.
  5. If $\mathsf{ct}_{msg} \in \mathcal{R}$, query $U_{\mathsf{msg}^*}(P)$ and output the response.

In **Hybrid** 3.1, the decryption of ct_{key} is used by $\mathcal{S}$ to decrypt ct_{msg} while running $\mathsf{HW.Run}(\mathsf{hdl}_P, \cdot)$. This ct_{key} is a valid encryption of $\mathsf{sk}_{\mathsf{pke}}$ because **Hybrid** 3.0 and

Hybrid 3.1 ensure that the encryption of $\mathsf{sk}_{\mathsf{pke}}$ sent from KME to DE and then the one from DE to FE both reach FE unmodified. Hence, the $\mathsf{sk}_{\mathsf{pke}}$ got by decrypting ct_{msg} is same as the one from msk. Thus, **Hybrid** 4 is indistinguishable from **Hybrid** 3.1 for any $\mathsf{ct}_{msg} \notin \mathcal{R}$. Now, let us consider the case of $\mathsf{ct}_{msg} \in \mathcal{R}$. $\mathcal{S}$ has the restriction that it can use the U_{msg^*} oracle only for a P for which $\mathsf{tag}_P \in \mathcal{K}$. From **Hybrid** 3.1, we know that $\mathsf{HW.Run}(\mathsf{hdl}_P, \cdot)$ does not output $\perp$ only when run with a valid $\mathsf{report}_{sk} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{DE}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}), \mathsf{mac})$ which is output by a DE "provision" query. Hence, sig is a valid signature of the tag_P contained in report. Also, $\mathsf{tag}_P \in \mathcal{K}$ with the honest bit set, as ensured in **Hybrid** 2. So, when a $\mathsf{HW.Run}$ "run" query is made for hdl_P, $\mathcal{S}$ is allowed use its U_{msg^*} oracle to output the $\mathsf{FE.Dec}$ result. Thus, **Hybrid** 4 is indistinguishable from **Hybrid** 3.1 for any ct_{msg}.

The following set of hybrids will help $\mathcal{S}$ replace an encryption of $\mathsf{sk}_{\mathsf{pke}}$ with an encryption of zeros. In order to prove the indistinguishability, we will argue that all the FE algorithms run independent of the $\mathsf{sk}_{\mathsf{pke}}$ encrypted in ct_{sk}, and that $\mathcal{A}$ does not get any information about the value encrypted in ct_{sk}.

Hybrid 5.0 As in **Hybrid** 4, except that $\mathcal{S}$ maintains a list L_{DE2} of all $(\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}})$ that are part of $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{DE}, \text{"init setup"}, (\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}}), \sigma)$ output by $\mathsf{HW.Run\&Quote}(\mathsf{hdl}_{DE}, \text{"init setup"}, \cdot)$ for $\mathsf{hdl}_{DE} \in \Psi$. And now, when $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, \text{"provision"}, \mathsf{quote}, \mathsf{params})$ is called $\mathcal{S}$ outputs $\perp$ when $(\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}}) \notin L_{DE2}$.

The Remote Attestation security ensures that $\mathcal{A}$ can provide a fake quote on a $\mathsf{pk}_{\mathsf{ra}}$ not provided by DE only with negligible probability (Lemma 4.7.5). Thus ensures that KME provides an encryption of $\mathsf{sk}_{\mathsf{pke}}$ only under a public key $\mathsf{pk}_{\mathsf{ra}}$ generated inside $Q_{DE} \in \Psi$ i.e., when $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, \text{" provision"}, \mathsf{quote}, \mathsf{params})$ is called with a valid quote output by a valid instance of DE.

Hybrid 5.1 As in **Hybrid** 5.0, except that $\mathcal{S}$ maintains a list L_{FE} of all $(\mathsf{sid}, \mathsf{pk}_{\mathsf{la}})$ that are part of $\mathsf{report} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_P, (\text{"init"}, \mathsf{sid}, \mathsf{pk}_{\mathsf{la}}), \mathsf{mac})$ output by $\mathsf{HW.Run\&Report}(\mathsf{hdl}_P, \text{"init"}, \cdot)$ for $\mathsf{hdl}_P \in \mathcal{K}$ with the honest bit set. And when $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \text{"provision"}, \mathsf{report}, \mathsf{sig})$ is called for a $\mathsf{hdl}_{DE} \in \Psi$, $\mathcal{S}$ outputs $\perp$ when report contains $\mathsf{tag}_P \in \mathcal{K}$ but $(\mathsf{sid}, \mathsf{pk}_{\mathsf{la}}) \notin L_{FE}$.

This is ensured by the Local Attestation security (Lemma 4.7.6). And, this shows that Q_{DE} only outputs $\mathsf{sk}_{\mathsf{pke}}$ encrypted under some $\mathsf{pk}_{\mathsf{la}}$ that was generated by a $Q_{FE}(\mathsf{hdl}_P, \cdot)$

running a program P that has been queried to $\mathsf{FE.Keygen}$.

Hybrid 5.2 As in **Hybrid** 5.1, except that when the KM oracle calls $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"provision"}, \cdot, \cdot))$, $\mathcal{S}$ replaces ct_{sk} in the output with $\mathsf{PKE.Enc}(0^{|\mathsf{sk}_{\mathsf{pke}}|})$.

Lemma 4.7.5 and Lemma 4.7.6 ensure that $\mathsf{sk}_{\mathsf{pke}}$ is encrypted only under $\mathsf{pk}_{\mathsf{ra}}$ and $\mathsf{pk}_{\mathsf{la}}$ generated by valid enclaves and $\mathcal{A}$ has no access to the corresponding secret keys. Now, Lemma 4.7.7 will use the IND-CCA2 security gameto argue that $\mathcal{A}$ cannot distinguish whether ct_{sk} has an encryption of zeros or $\mathsf{sk}_{\mathsf{pke}}$ under $\mathsf{pk}_{\mathsf{ra}}$ of the DE, and whether ct_{key} is an encryption of zeros or $\mathsf{sk}_{\mathsf{pke}}$ under $\mathsf{pk}_{\mathsf{la}}$ of a valid FE.

Hybrid 6 As in **Hybrid** 5.2, except that $\mathsf{FE.Enc}^*$ is used instead of $\mathsf{FE.Enc}$.

We are now ready to use the IND-CCA2 security property of PKE to replace ct_{msg} which was an encryption of msg) with an encryption of zeros, as shown in Lemma 4.7.8.

Security proof lemmata

Lemma 4.7.2. *If the signature scheme* S *is existentially unforgeable as in Definition 4.6.4, then* **Hybrid** 2 *is indistinguishable from* **Hybrid** 1*.*

Proof. Let $\mathcal{A}$ be an adversary which distinguishes between **Hybrid** 1 and **Hybrid** 2. We will use it to break the EUF-CMA security of S. We will get a verification key $\mathsf{vk}^*_{\mathsf{sign}}$ and an access to $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ oracle from the EUF-CMA challenger. $\mathcal{S}$ sets this $\mathsf{vk}^*_{\mathsf{sign}}$ as part of the mpk. Whenever $\mathcal{S}$ has to sign a message using $\mathsf{sk}^*_{\mathsf{sign}}$, it uses the $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ oracle. Also, our construction does not ever need a direct access to $\mathsf{sk}^*_{\mathsf{sign}}$; it is used only to sign messages for which the oracle provided by the challenger can be used. Now, if $\mathcal{A}$ can distinguish between the two hybrids, as we argued earlier, it is only because $\mathcal{A}$ makes a "provision" query to the $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \cdot)$ oracle with a $\mathsf{hdl}_{DE} \in \Psi$ that has $\mathsf{vk}^*_{\mathsf{sign}}$ in its *setupstate* and with a valid signature sig on a $\mathsf{tag}_P \notin \mathcal{K}$. We will output $(\mathsf{tag}_P, \mathsf{sig})$ as our forgery to the EUF-CMA challenger. □

Lemma 4.7.3. *If the signature scheme* S *is existentially unforgeable as in Definition 4.6.4, then* **Hybrid** 3.0 *is indistinguishable from* **Hybrid** 2*.*

Proof. Let $\mathcal{A}$ be an adversary which distinguishes between **Hybrid** 2 and **Hybrid** 3.0. We will use it to break the EUF-CMA security of S. We will get a verification key $\mathsf{vk}^*_{\mathsf{sign}}$ and an access to $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ oracle from the EUF-CMA challenger. $\mathcal{S}$ sets this $\mathsf{vk}^*_{\mathsf{sign}}$ as part of the mpk. Whenever $\mathcal{S}$ has to sign a message with $\mathsf{sk}^*_{\mathsf{sign}}$, it uses the $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ oracle. As mentioned in the proof of Lemma 4.7.2, $\mathcal{S}$ never needs a direct access to $\mathsf{sk}^*_{\mathsf{sign}}$. Now, if $\mathcal{A}$ can distinguish between the two hybrids, as we argued earlier, it is only because $\mathcal{A}$ makes a "complete setup" query to the $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \cdot)$ oracle with a valid signature σ_{sk} for $(\mathsf{sid}, \mathsf{ct}_{sk}) \notin L_{KM}$ but sid has an entry in *setupstate*. Also, $\mathsf{hdl}_{DE} \in \Psi$ and hence has $\mathsf{vk}^*_{\mathsf{sign}}$ in its *setupstate*. We will output $((\mathsf{sid}, \mathsf{ct}_{sk}), \sigma_{sk})$ as our forgery to the EUF-CMA challenger. □

Lemma 4.7.4. *If the Local Attestation process of* HW *is secure as in Definition 3.3.1, then* **Hybrid** 3.1 *is indistinguishable from* **Hybrid** 3.0.

The proof of this lemma is similar to Lemma 4.7.3, since $\mathsf{sk}_{\mathsf{report}}$ is not used by $\mathcal{S}$ other than to produce a report.

Lemma 4.7.5. *If Remote Attestation is secure as in Definition 3.3.2, then* **Hybrid** 5.0 *is indistinguishable from* **Hybrid** 4.

The proof of this lemma is similar to Lemma 4.7.3 since $\mathsf{sk}_{\mathsf{quote}}$ is not used by $\mathcal{S}$ except for producing a quote.

Lemma 4.7.6. *If Local Attestation is secure as in Definition 3.3.1, then* **Hybrid** 5.1 *is indistinguishable from* **Hybrid** 5.0.

The proof of this lemma is again similar to Lemma 4.7.3 since $\mathsf{sk}_{\mathsf{report}}$ is not used by $\mathcal{S}$ except for producing a report.

Lemma 4.7.7. *If* PKE *is an IND-CCA2 secure encryption scheme, then* **Hybrid** 5.2 *is indistinguishable from* **Hybrid** 5.1.

Proof. We will run two IND-CCA2 games in parallel, one for ct_{sk} and another for ct_{key}. It can be easily shown that this variant is equivalent to the regular IND-CCA2 security game. The IND-CCA2 challenger provides two challenge public keys pk^*_1 and pk^*_2. $\mathcal{S}$ sets $\mathsf{pk}_{\mathsf{ra}} = \mathsf{pk}^*_1$ and $\mathsf{pk}_{\mathsf{la}} = \mathsf{pk}^*_2$. Now, $\{\mathsf{sk}_{\mathsf{pke}}, 0^{|\mathsf{sk}_{\mathsf{pke}}|}\}$ is provided as the challenge message pair for both the games. The challenger returns ct^*_1 and ct^*_2, which are encryptions of either the left messages or the right messages from the each pair. Note that we use the same challenge bit for both the games. $\mathcal{S}$ sets $\mathsf{ct}_{sk} = \mathsf{ct}^*_1$ and $\mathsf{ct}_{key} = \mathsf{ct}^*_2$.

Now we argue that when the left messages are encrypted, the view of $\mathcal{A}$ is equivalent to **Hybrid** 5.1, and when the right messages are encrypted, the view is equivalent to **Hybrid** 5.2. This is because the other information that $\mathcal{A}$ gets do not depend on the value encoded in ct_{sk} or ct_{key}. We argue this as follows. We have already established that $\mathcal{A}$ only gets ct_{sk} encrypted with a $\mathsf{pk}_{\mathsf{ra}}$ generated in DE from KME. Similarly, $\mathcal{A}$ only gets ct_{key} encrypted with a $\mathsf{pk}_{\mathsf{la}}$ generated in a valid FE from DE. In addition to these, when interacting with messages from a valid Q_{DE} or $Q_{FE}(\cdot)$, $\mathcal{S}$ either uses the $\mathsf{sk}_{\mathsf{pke}}$ from msk or the U_{msg} oracle to answer the queries and not the decryption of ct_{key}.

Hence, when $\mathcal{A}$ decides between the two hybrids we forward the corresponding answer to the IND-CCA2 challenger. If $\mathcal{A}$ can distinguish between these two hybrids with non-negligible probability, then the IND-CCA2 security of PKE can be broken with non-negligible probability. □

Lemma 4.7.8. *If* PKE *is an IND-CCA2 secure encryption scheme, then* **Hybrid** 6 *is indistinguishable from* **Hybrid** 5.2.

Proof. The IND-CCA2 challenger provides the challenge public key pk^*. During FE.Setup $\mathcal{S}$ sets $\mathsf{pk}_{\mathsf{pke}} = \mathsf{pk}^*$. Now, msg and $0^{|\mathsf{msg}|}$ are provided as the challenge messages. The challenger returns ct^*, which is an encryption of either of those with equal probability. $\mathcal{S}$ sets $\mathsf{ct}_{msg} = \mathsf{ct}^*$. When $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{“run”}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ is called with a valid report_{sk} to $\mathsf{hdl}_P \in \mathcal{K}$ with the honest bit set, $\mathcal{S}$ uses the U_{msg^*} oracle for a challenge ciphertext $\mathsf{ct}_{msg} \in \mathcal{R}$ from **Hybrid** 4. Now, for any $\mathsf{ct}_{msg} \notin \mathcal{R}$, $\mathcal{S}$ neither has the oracles nor has the sk^* corresponding to pk^* in msk. But, the decryption oracle provided by the IND-CCA2 challenger can be used for any $\mathsf{ct}_{msg} \notin \mathcal{R}$. Hence, $\mathcal{S}$ can answer all the $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{“run”}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ queries. Thus, the view of $\mathcal{A}$ is identical to **Hybrid** 5 when msg is encrypted in ct^* and **Hybrid** 6 when zeros are encrypted in ct^*. So we can forward the answer corresponding to $\mathcal{A}$'s answer to the IND-CCA2 challenger. If $\mathcal{A}$ can distinguish between these two hybrids with non-negligible probability, the IND-CCA2 security of PKE can be broken with non-negligible probability. □

4.8 FE construction in the stronger security model

We will now present the formal description of our second FE construction which can be proven secure in the stronger security models of HW and FE described in Definition 4.6.2. This definition does not let the simulator simulate the HW oracle in the ideal world.

The trusted authority platform TA and decryption node platform DN each have access to instances of HW. We assume $\mathsf{HW.Setup}(1^\lambda)$ has been called for each of these instances before they are used in the protocol and the output params was recorded. Let PKE denote an IND-CCA2 secure public key encryption scheme (Definition 4.6.5) with the weak robustness property[10], let S denote an existentially unforgeable signature scheme (Definition 4.6.4) and E denote an IND-CPA secure secret key encryption scheme (Definition 4.6.3).

FE.Setup(1^λ) The key manager enclave program Q_{KME} is defined in Figure 4.9. Let state denote an internal state variable.

1. Run $\mathsf{hdl}_{KME} \leftarrow \mathsf{HW.Load}(\mathsf{params}, Q_{KME})$.
2. Run $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{vk}_{\mathsf{sign}}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"init"}, 1^\lambda))$.
3. Output the master public key $\mathsf{mpk} := \mathsf{pk}_{\mathsf{pke}}$ and the master secret key $\mathsf{msk} := \mathsf{hdl}_{KME}$.

FE.Keygen(msk, P)

1. Parse msk as a handle to $\mathsf{HW.Run}$.
2. Derive tag_P and call $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"sign"}, \mathsf{tag}_P))$.
3. Output $\mathsf{sk}_p := \mathsf{sig}$.

FE.Enc$(\mathsf{mpk}, \mathsf{msg})$

1. Parse $\mathsf{mpk} = (\mathsf{pk}, \mathsf{vk})$.
2. Sample an ephemeral key $\mathsf{ek} \leftarrow \mathsf{E.KeyGen}(1^\lambda)$ and use it to encrypt the message $\mathsf{ct}_m \leftarrow \mathsf{E.Enc}(\mathsf{ek}, \mathsf{msg})$.
3. Encrypt the ephemeral key under pk along with the hash of ct_m: $\mathsf{ct}_k \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, [\mathsf{ek}, H(\mathsf{ct}_m)])$.
4. Output $\mathsf{ct} := (\mathsf{ct}_k, \mathsf{ct}_m)$.

[10] We actually need one PKE scheme with IND-CPA security and weak robustness property and another PKE scheme with IND-CCA2 security

Figure 4.9: Q_{KME} II

- On input ("init", 1^λ):
  1. Run $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ and $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)$
  2. Update state to $(\mathsf{sk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{sign}}, \mathsf{vk}_{\mathsf{sign}})$ and output $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{vk}_{\mathsf{sign}})$
- On input ("provision", quote, params):
  1. Parse $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_P, \mathsf{input}, \mathsf{output}, \sigma)$, and parse $\mathsf{output} = (\mathsf{sid}, \mathsf{pk}^1, \mathsf{pk}^2, \mathsf{sk}_P, \mathsf{ct}_k)$.
  2. Run $b \leftarrow \mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$ on quote. If $b = 1$, retrieve $\mathsf{sk}_{\mathsf{pke}}$ and $\mathsf{vk}_{\mathsf{sign}}$ from state. If $b = 0$ output $\perp$.
  3. Run $b \leftarrow \mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_P, \mathsf{tag}_P)$. If $b = 0$, output $\perp$.
  4. Run $(\mathsf{ek}, h) \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_k)$
  5. Compute $\mathsf{ct}^1_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}^1, \mathsf{ek}||\mathsf{vk}_{\mathsf{sign}})$ and $\mathsf{ct}^2_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}^2, \mathsf{ek}||\mathsf{vk}_{\mathsf{sign}})$
  6. Compute $\sigma_{sk} = \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, (\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h))$ and output $(\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h, \sigma_{sk})$.
- On input ("sign", msg):
  Compute $\mathsf{sig} \leftarrow \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{msg})$ and output sig.

FE.Dec$^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{sk}_P, \mathsf{ct})$ The decryption enclave program Q_{DE} parametrized by P is defined in Figure 4.10. The security parameter λ is hardcoded into the program. The Q_{DE} here can be seen as the merge of the Q_{DE} and Q_{FE} in our first construction.

1. Run $\mathsf{hdl}_{DE} \leftarrow \mathsf{HW.Load}(\mathsf{params}, Q_{DE})$.
2. Call $\mathsf{quote} \leftarrow \mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}_{DE}, \text{"init dec"}, \mathsf{sk}_P, \mathsf{ct}_k)$.
3. Query KM(quote), which internally runs $(\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h, \sigma_{sk}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"provision"}, \mathsf{quote}, \mathsf{params}))$[11].

[11] We could again use HW.Run&Quote here instead of explicitly creating the signature σ_k. If we do that, the verification step in DE would involve using the Intel Attestation Service.

Figure 4.10: Q_{DE} II

- On input ("init dec", $\mathsf{sk}_P, \mathsf{ct}_k$):
  1. Run $\mathsf{PKE.KeyGen}(1^\lambda)$ twice to get $(\mathsf{pk}^1_{\mathsf{ra}}, \mathsf{sk}^1_{\mathsf{ra}})$ and $(\mathsf{pk}^2_{\mathsf{ra}}, \mathsf{sk}^2_{\mathsf{ra}})$.
  2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
  3. Update state to $(\mathsf{sid}, \mathsf{sk}^1_{\mathsf{ra}}, \mathsf{sk}^2_{\mathsf{ra}})$, and output $(\mathsf{sid}, \mathsf{pk}^1_{\mathsf{ra}}, \mathsf{pk}^2_{\mathsf{ra}}, \mathsf{sk}_P, \mathsf{ct}_k)$.
- On input ("complete dec", $(\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h), \sigma_{sk}$):
  1. Look up the state to obtain the entry $(\mathsf{sid}, \mathsf{sk}^1_{\mathsf{ra}}, \mathsf{sk}^2_{\mathsf{ra}})$. If no entry exists for sid, output $\perp$.
  2. Verify the signature $b \leftarrow \mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}}, \sigma_k, (\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h))$. If $b = 0$, output $\perp$.
  3. Check that $h = H(\mathsf{ct}_m)$. If not, output $\perp$.
  4. Decrypt $m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}^1_{\mathsf{ra}}, \mathsf{ct}^1_{sk})$.
  5. If $m = \perp$, decrypt and output $\mathsf{output} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}^2_{\mathsf{ra}}, \mathsf{ct}^2_{sk})$.
  6. Parse $m = (\mathsf{ek}, \mathsf{vk}_{\mathsf{sign}})$ and compute $x \leftarrow \mathsf{E.Dec}(\mathsf{ek}, \mathsf{ct}_m)$.
  7. Run P on x and output $\mathsf{output} := P(x)$.

4. Call $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, (\text{"complete dec"}, \mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h, \sigma_{sk}))$.
5. Output its result output.

4.8.1 Security overview

Theorem 4.8.1. *If* E *is an IND-CPA secret key encryption scheme,* S *is an EUF-CMA secure signature scheme,* PKE *is an IND-CCA2 secure public key encryption scheme with weak robustness property and* HW *is a secure hardware scheme, then* FE *is a secure functional encryption scheme according to Definition 4.6.2.*

We will mention here some of the challenges faced while proving the security of our construction. The main difference from the proof of our first construction is that the HW algorithms are not simulated but are run as in the the real world. Hence, when we use the

IND-CCA2 security of PKE to prove that the adversary does not learn any information from the communication between the enclaves, the decryption enclave will not have the correct secret key to decrypt the PKE ciphertext and hence cannot proceed to generate the correct output. To remedy that situation, DE sends two public keys and KME sends two ciphertexts during that step so that when the IND-CCA2 game is run for one of ciphertexts, the other ciphertext can be decrypted by DE to satisfy the correctness of the FE scheme. During this step, we will also use the indistinguishability of ciphertexts when the same messages are encrypted under different public keys. Also during this step, to help the programs decide whether the message got after decryption is correct or not, we require the robustness property from our PKE scheme which ensures that decryption outputs $\perp$ when a ciphertext is decrypted with a "wrong" key.

Discussion

- This construction can be modified to work like the first construction, where the decryption enclave is separated from the function enclave written by the user programmer.
- This construction allows us to achieve the stronger security notions of FE and HW. But, one might wonder how our KM oracle compares with the notion of hardware tokens in [72]. With an "oracle" being necessary due to the FE impossibility results, we made the functionality of the KM oracle minimal. In our construction, KM performs minimal crypto functionality: basic signing/encryption. (And it is an independent enclave DE without access to msk which runs the user-specified programs on user-specified inputs). Hence, it is relatively easier to implement the KM functionality secure against side-channels, when compared to the powerful hardware tokens. Also from a theoretical perspective, KM runs in time independent of the runtime of program and the length of msg, in contrast to the hardware tokens whose runtime depends on both the program and msg.
- The similarity of C-FE with our notion is that there is an "authority" mediating every decryption. If mediation by KM were a concern to an application of FE, the message sent by DE to the KME can be encrypted and anonymous communication mechanisms like Tor can be used to communicate to KM so that KM cannot discriminate against specific decryptor nodes (also helped by remote attestation using blind signatures). Also, our construction could be modified to achieve C-FE when the efficiency constraints are relaxed for the authority oracle such that they run in time independent on the length of the input but dependent on the function description

length. The construction in [168] requires the authority to run in time proportional to the length of function description and input.

4.9 Extensions and Future Work

Private Key MIFE There is a private key variant of MIFE where producing a valid ciphertext for the ith input to a function requires a secret encryption key ek_i. Invoking the decryption algorithm on inputs produced with an invalid key does not reveal any information about the plaintext data. For some multi-input functionalities, private key MIFE is necessary to achieve meaningful security. For example, consider the order function $ord(x, y) = 1$ iff $x > y$. In the public key setting, given an encryption c_x of x and a functional key for ord the decryptor can produce valid ciphertexts for any arbitrary integer y in order to learn $ord(x, y)$, and can recover x by binary search. IRON supports private key MIFE. In this mode, the Authority appends a signature on the appropriate index to the public encryption key, i.e. $ek_i = sig_i || \mathsf{pk}_{\mathsf{pke}}$ where sig_i is a signature on the integer i using $\mathsf{sk}_{\mathsf{sign}}$. To encrypt a message m with ek_i, the encryptor uses $\mathsf{pk}_{\mathsf{pke}}$ to produce a public key encryption $c_{i,m}$ of $sig_i || m$. When an enclave on the decryption node receives $c_{i,m}$ as the ith input to a function, it uses $\mathsf{sk}_{\mathsf{pke}}$ to decrypt $c_{i,m}$ and validates the signature appended to the message using $\mathsf{vk}_{\mathsf{sign}}$. If this is not a valid signature on the index i then the enclave aborts the operation, and otherwise it proceeds with m.

Function Private FE Currently, IRON supports a version of FE where the function to be evaluated is not hidden from the decryptor, and moreover, it is not hidden from the decryption node. Function private FE [50] could be supported by running a single enclave on the decryption node that receives encrypted and signed function code, decrypts the function code, checks the signature, and executes the decrypted code either through an interpreter or by writing the code to pre-allocated WX enabled pages. However, doing this securely would require the capability of full program obfuscation in SGX. It has not yet been demonstrated that this is possible to achieve *practically* for generic programs given the current side-channel attacks on SGX, though some effort in this direction was made in [185] and demonstrated on SGX-like special purpose hardware in [170].

Multi-Authority FE In multi-authority FE [66], the trust is distributed among multiple authorities instead of having a single authority manage all the credentials. Clients must obtain secret keys from all (or a suitably large subset) of the authorities in order to be able

to decrypt ciphertexts. Since the secret keys in Iron are simply signatures, it would be easy to augment Iron to support this feature by using threshold-signatures and multiple KMEs.

Chapter 5

StealthDB: A Scalable Encrypted Database on SGX

5.1 Introduction

In this chapter, we present our study on designing a secure version of a real-world system using Intel SGX, with transactional databases.

Transactional database systems are designed to store and process enterprise data with ACID (Atomicity, Consistency, Isolation, Durability) guarantees. A lot of enterprises now use third party public cloud infrastructure or service providers like AWS, Microsoft Azure and Google Cloud to maintain their databases. This places complete trust on their data with these providers, as we discussed earlier.

To tackle this problem, research has been done to build "encryption-in-use" mechanisms that greatly improve security by preventing the attackers and even the cloud operators from ever seeing the data in clear. A lot of work has been done on improving the security and performance on a subset of SQL operations as systematized in the survey by [90], but only a handful of systems are complete and evaluated at scale. The state of art encryption-in-use database systems which have been evaluated at scale can be divided into two main categories:

(A) systems built using advanced encryption schemes that allow to perform operations over the ciphertexts [182, 181, 177], and

(B) systems that leverage a trusted processing device (e.g., FPGA, IBM secure co-processor) to perform operations [20, 26, 85].

A practical encrypted database design is evaluated in terms of the following four aspects:

- security: leakage profile and security assumptions. Leakage profile characterizes the amount of data leakage introduced by the design. Security assumptions include the mathematical assumptions for the cryptography and the trusted computing base (TCB) and other trust assumptions for the trusted hardware.
- functionality: the SQL operations and DBMS functions supported.
- performance: throughput, latency and scalability to large datasets.
- intrusiveness level: amount of changes to the underlying DBMS.

CryptDB [182] is a seminal work in this area using property-preserving encryption schemes to execute queries over encrypted data. But, these schemes do not offer strong security and when used in multiple columns they are found to leak extensive information for real-world datasets [169, 118]. Also, [182] requires extensive computations (re-encryption of entire columns) on a trusted proxy or the client to support all the SQL queries. The other systems using advanced encryption schemes either have a very limited functionality [177, 63, 133] or incur heavy computational and storage overheads [181].

Cipherbase [20] offers a scalable design for transactional workloads with a strong leakage profile and complete SQL support, by leveraging on trusted hardware. But, the system uses FPGAs as its trusted hardware and hence has the following security implications: (i) an initial trusted and on-premise key loading phase is required for every FPGA device used, (ii) a huge trust is placed on the FPGA"shell" layer [17] implemented by the cloud operators which monitors the user operations on the FPGA to ensure the safety of the device. As such, significant research is required to use FPGAs as *trusted* hardware in cloud-based applications. The other trusted hardware based systems [26, 85] offer improved leakage profile but only at the cost of extensive DBMS changes, much larger TCB and huge performance overheads for large transactional workloads.

In this work, we study how to build an encrypted database system from a standard CPU leveraging the Intel Software Guard Extensions (SGX) instruction set [159]. SGX enables the creation of a small encrypted memory container (enclave) that can be accessed only by a predefined trusted code. The content of the enclave is protected from untrusted applications and even the system administrators, OS and hypervisor. Also, SGX is available in all the recent and future releases of Intel CPUs Hence, SGX offers a great direction for protecting applications in cloud environments. But, SGX has its own set of restrictions. It requires rewriting of applications by partitioning code into trusted and untrusted segments.

Also, there is a 90 MB bound on "secure" memory to run the trusted enclave code, which is not nearly enough for even medium size database workloads.[1] Additionally, SGX is vulnerable to memory, cache and other side-channel leakages, lacks syscalls and IO support, and incurs high overheads for switching between enclave and non-enclave modes, which further limit the complexity and functionality of the trusted enclave code. As such, one cannot take a DBMS system and naively try to "run it in an enclave". But, it is important for an encrypted database design to get around these limitations without having to make extensive changes to the underlying DBMS, while still achieving the performance, security and functionality goals. Also, it is not clear whether a design that works well for another trusted hardware can be ported to SGX while preserving the end-to-end security guarantees, since each hardware has its unique set of security and usability requirements.

5.1.1 Our contributions

Design choices with SGX We first investigate three possible design choices for an encrypted database with SGX in Section 5.4.2 by varying the DBMS components run inside an enclave. Through a set of benchmarking experiments, we identify a design that works best for our design goals (Section 5.5). We develop on that to get the StealthDB design.

StealthDB The StealthDB system provides a complete SQL support, strong end to end security guarantees and performance with very minimal changes to the underlying DBMS. A high-level overview of our system is presented in Figure 5.1. StealthDB uses AES-GCM, a semantically secure encryption scheme to encrypt all the data items in the database. During query execution, client encrypts the query string and sends the ciphertext to the server. We implement a query parser inside an enclave, which first decrypts the ciphertext to get the query and parses the query to output a version with all the constants encrypted. For example, when a client sends ENC(select * from item where name = 'John'), it is converted to select * from item where name = 'ENC(John)' by our enclave parser. To support queries of this form, we define encrypted datatypes and implement the operators over these datatypes inside an enclave. We make the operators data-oblivious [173] to protect against SGX side-channel attacks. We also encrypt the index file pages before they are written to disk. These changes are not intrusive and hence enable StealthDB inherit the functionality of the underlying DBMS completely.

[1]Although various SGX extensions are promised by Intel in future releases with larger secure memory, they are not available in the market yet and unclear when they will be. We also argue later that these extensions should not affect our conclusions on the architecture of an encrypted DBMS with SGX.

Security StealthDB offers a stronger leakage profile compared to the prior complete encrypted database systems. A snapshot adversary [187, 91, 55, 27, 78] learns only the "shape" of the database which includes the dimensions of the data structures maintained by the DBMS, along the recently collected query log information. An adversary with persistent access to memory and disk learns the inequalities (¡, ¿, =) between the encrypted values in the indexes which are compared during the query execution, along with the query access pattern which includes the position of the result records in the database. In general, the enclave code can be thought of as providing a black-box access to the DBMS to perform the computations on encrypted data values and obtain the output (encrypted or unencrypted depending on the specification), without leaking any other information about the input data values. We explain our leakage profile in more detail in Section 5.6, and this profile matches the state-of-art (the strongest version in [20]) when providing either reasonable performance[2] or intrusiveness levels for large transactional workloads. Also, our TCB just includes the processor, the enclave code along with the SGX hardware and the attestation procedure. Our clients use the SGX attestation procedure to attest the correctness of the enclave code before issuing queries. This combined with the simplicity of the enclave code reduces the trust to be placed on the enclave code.

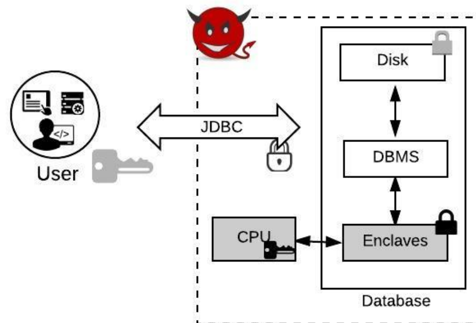


Figure 5.1: High-level architecture overview of StealthDB

Evaluation We implement our design on top of an existing Postgres DBMS. Our new encrypted datatypes and the corresponding primitive operations (UDFs) are added as extensions in Postegres [183]. The only component that needs modifying the Postgres code is to encrypt/decrypt the index files when they are stored to/accessed from disk

[2]From our experience talking to the industry on the possible adoption of StealthDB, 50% to 2× overhead in *performance* is a reasonable penalty for the benefit of security against untrusted cloud operators.

and this just needs a three lines change in the Postgres codebase. None of these changes are intrusive, or specific to Postgres. Hence, this design principle lets StealthDB benefit directly from any performance or feature improvements to the underlying DBMS engine. Performance-wise, StealthDB scales to large datasets with a similar complexity to an unmodified DBMS engine working on unencrypted data, adding only a constant overhead for each query. Our evaluation results in Section 5.8 show that the system can process transactional (OLTP) queries with a 30% reduction in throughput and ≈ 1 ms overhead in latency over an unencrypted DBMS with > 10M total rows (or 2 GB plaintext) of a TPC-C warehouse database, a standard benchmark for transactional queries [1], for scale factor $W = 16$.

5.2 Intel SGX

There are three main functionalities that SGX enclaves achieve: isolation, sealing and attestation. We discussed these in detail in Chapters 3 and 4. In addition, we use key establishment procedure built on top of the attestation procedures. We brief that here.

Key establishment during attestation. Key establishment between two enclaves or between an enclave and a remote party can be accomplished on top of the local/remote attestation process. An enclave can send the key shares (for eg., a Diffie-Hellman key share g^a) and include them as the *additional authentication data* to MAC. Thus attestation provides authenticity and integrity to the key share from the enclave. In our system, we will very often run the key establishment phase on top of local/remote attestation to establish a secure channel for communication between two enclaves or between an enclave and a remote party using the established shared secret key.

5.3 Platform Overview

5.3.1 Usage Model.

We work with the following setting. A data owner aims to store and process data securely on a remote untrusted SQL database server. She authorizes clients by issuing them *credentials*, and wants to support the authorized clients to issue queries to the server. The server maintains a *credential database* for the authorized clients in an encrypted form. Each client authenticates to the server using its credentials, which will enable the client to issue

its permitted queries to the database. The server in our model is equipped with a secure processor, such as Intel SGX. Hence, the server can be identified with some "platform-key" established by Intel SGX. The data owner and clients engage in the attestation of SGX enclaves in the server and on successful attestations, transfer any secret or sensitive material (master key, credentials, queries, etc.) to those enclaves via secure channels.

5.3.2 Threat Model

StealthDB provides security against passive adversaries. A passive adversary does not inject malicious code or alter the program execution in any way. But, it can read the contents of the memory, disk and all the communication, and hence may *passively* attempt to learn additional information from the data they observe.

There are two dimensions in which we analyze the threat model for our system. The first dimension is about the extent of access: adversaries restricted to monitoring the disk accesses versus the adversaries monitoring both the memory and disk accesses in the system. The second dimension is about the duration: adversaries getting snapshot accesses to memory or disk versus the much stronger ones which get persistent access to memory and disk. A snapshot attack might be due to a memory dump or some cold-boot attack by a malicious cloud provider or by a co-located client running on the same cloud server as the victim process which gets occasional access to the memory of the entire system due to access control bugs. SQL injection attacks [83, 121, 78], VM attack leaks [187, 91, 55, 27], disk theft and a "smash-and-grab" after a full system compromise [78] are some real-world examples of snapshot attacks [119].

5.4 Designing an Encrypted DB

In this section, we describe a few design goals we set out to achieve for our system. Then, we discuss and experiment with a few possible design choices possible when building an encrypted database from SGX.

5.4.1 Design Goals

The focus of StealthDB is on building a scalable encrypted database system that can support arbitrary query types, with a reasonable leakage. Construction of an encrypted

DBMS with a complete SQL support under any meaningful notion of security is an uphill task in this world where the proposed attacks [118, 119, 142, 117] completely dismantle the security of even the constructions with limited functionality (like searchable encryption) which had, what was thought to be, extremely minimal leakage (reveal just the locations of the results of each query). There has been extensive research to secure subset of SQL operations, but they have barely made their way into a real world DBMS unless a design for a complete system is provided. For instance, the CryptDB design was part of or inspired many real-world systems [182, 164, 114, 105] due to an almost complete DBMS support. In this regard, we set our design goals as follows:

- Functionality goal: complete support to the SQL functionality of the underlying DBMS.
- Non-intrusiveness goal: minor modifications to the core DBMS operations of the underlying DBMS, for the encrypted database to retain the DBMS properties. If the underlying DBMS is ACID compliant, supports triggers and stored procedures, so should the encrypted database.
- Performance goal: high throughput and low latency when scaling to large datasets.
- Security goal: We will start by stating the security goals informally:
  - a snapshot adversary on both memory and disk should learn no information about the individual data items.
  - a persistent adversary on both memory and disk learns no information about the encrypted data that are not compared when the queries are processed, other than that they are not part of the query processing. Even for the data of the query execution, the leakage should match or be stronger than the previous works supporting complete SQL.

  We will later study the security for each proposed design. And, the leakage profile of the chosen StealthDB design will be detailed in Section 5.6.

There is an inherent trade-off here between security and performance which will influence our design choices. There is a lower bound of logarithmic overhead in performance [101, 64], just to support encrypted search without any leakage. This also translates to the trade-off between efficiency and the information leakage during the index building and usage. Moreover, we also aim to design secure versions of arithmetic and other operators to support SQL completely. Hence in this work, we lean towards achieving a good performance for

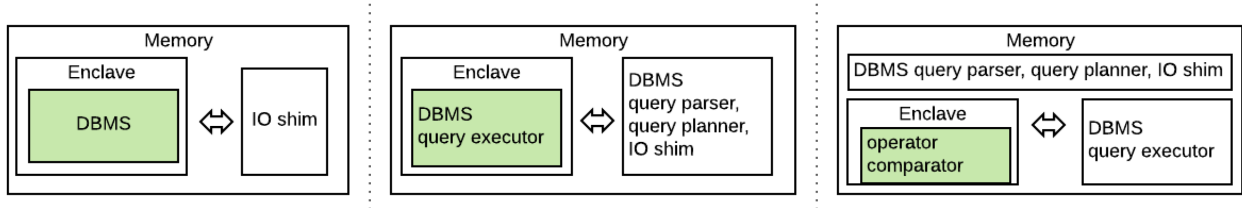


Figure 5.2: Three alternative design choices for an encrypted database with SGX.

large transactional workloads, while trying to achieve the best security possible for that performance.

5.4.2 Designing an Encrypted DB from SGX

We consider three design choices and evaluate them on a few micro experiments to help us understand how to build an encrypted database system with SGX. The design choices are summarized in Figure 5.2. We envision that in all three design choices data is encrypted on disk using a semantically secure encryption scheme. The designs differ in how queries are executed over the data.

The first, most obvious design would be to run the entire DBMS inside an enclave (left figure in 5.2). The data would be read from disk, decrypted transparently and then the DBMS would perform all necessary operations inside an enclave. However, SGX is not well suited for this task for a few reasons that we outlined earlier. The first issue is that SGX does not support IO or syscalls, so an additional outside shim layer would need to be exposed to talk to the kernel level, and the application dependencies need to be loaded inside (or outside via shim) an enclave. It is *feasible* to get around this issue using recent works such as Haven [32], Scone [21] and Graphene [202, 69]. They initiate the research in loading unmodified executables into enclaves with varying overheads. The second issue is that SGX is currently limited to 90 MB of working memory and significant penalties appear when going beyond that limit [174]. Future releases of SGX promise larger enclave sizes. However, the Merkle tree integrity protection for each memory page to prevent replay attacks does not scale well to larger enclaves. If these two issues were resolved, this design offers no information leakage to a snapshot attacker, since the memory used by the DBMS inside an enclave always remains encrypted. But, this design would keep a very large TCB inside the enclave: the entire DBMS engine, any communication logic with the "outside world" and dependencies. Since SGX is vulnerable to numerous side-channels,

very custom modifications to the DBMS are needed to prevent these attacks and make the code *oblivious*. Hence, this design is not very promising.

The second design we consider (middle figure in 5.2) keeps most of the DBMS in the untrusted zone. However, it places the query execution logic in the enclave. That is, when a query needs to be executed, individual tables can be brought into the enclave to perform selections, projections, joins, etc. The query plan, I/O and other DBMS parts remain in the untrusted memory. In terms of scalability, this design suffers from the same problems as the previous choice due to limited secure memory. Also, tables and indexes need to read from disk, deserialized and then loaded into enclave. In Figure 5.3 we show that the performance overhead for performing just this step (read and deserialize) inside an enclave is around 3× when the dataset fits within an enclave, and goes up to 9× for large datasets. In terms of security, the query processing logic would still need to do the non-trivial task of addressing the SGX side-channels. Finally, partitioning a DBMS to support this architecture is also a challenging task.

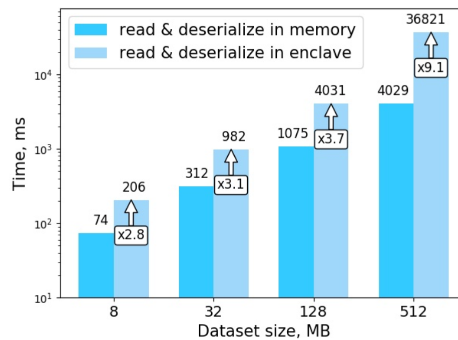
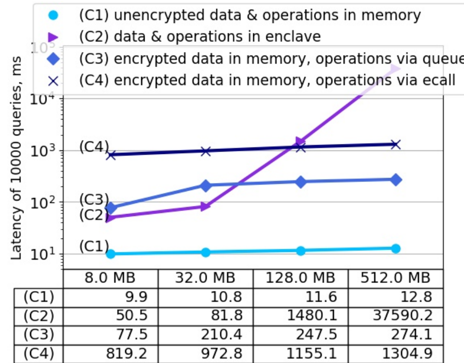


Figure 5.3: Initialization time comparing in memory and in enclave deserialization for different dataset sizes.

In the third design, we keep most of the DBMS in the untrusted zone, and the dataset would reside in the untrusted memory with the data items encrypted individually. At the lowest level of the parsed query tree, each query is eventually broken down into some *primitive* operators (e.g., $<=, >=, +, *$) over individual data values. To perform operations over encrypted data in this design, we transfer individual data item(s) to an enclave, followed by the decryption of input, the operator function and the encryption of output inside the enclave. The advantage of this design is that the communication with the disk and network layers would remain unchanged. Overall, minimal changes to the DBMS are needed – one only needs to change how primitive operators on data values are performed.



	8.0 MB	32.0 MB	128.0 MB	512.0 MB
(C1)	9.9	10.8	11.6	12.8
(C2)	50.5	81.8	1480.1	37590.2
(C3)	77.5	210.4	247.5	274.1
(C4)	819.2	972.8	1155.1	1304.9

Figure 5.4: Latency to execute random binary tree searches comparing different approaches.

Two different implementations of the partial approach: comparison function as trusted ecalls and the exit-less communication via a queue for transferring data to/from an enclave.

Also, the amount of code/data inside an enclave will remain a very small constant. This keeps the TCB very small, and it is easy to make it data-oblivious. Hence, we build on this design idea in Section 5.5. However, this design leaks relationship between encrypted data values during query execution in this design as discussed in Section 5.6.

In Figure 5.4, we compare performance of performing B-tree searches over database indexes in later two design choices. As expected, one can see that performing a search when an entire B-tree is loaded inside an enclave does not scale to larger datasets. (However, it performs well when the tree size is very small and can be fit entirely into an enclave.) In the third design, when the B-tree is kept encrypted in the untrusted memory but individual comparisons are executed in an enclave, we see up to 100× overheads compared to performing the search over unencrypted data. This can be explained by high switching costs for ecall/ocall functions, which are used for enclave entry/exit. Using an *exit-less* communication mechanism via a shared queue [174], we can reduce this overhead by $5\times -10\times$.

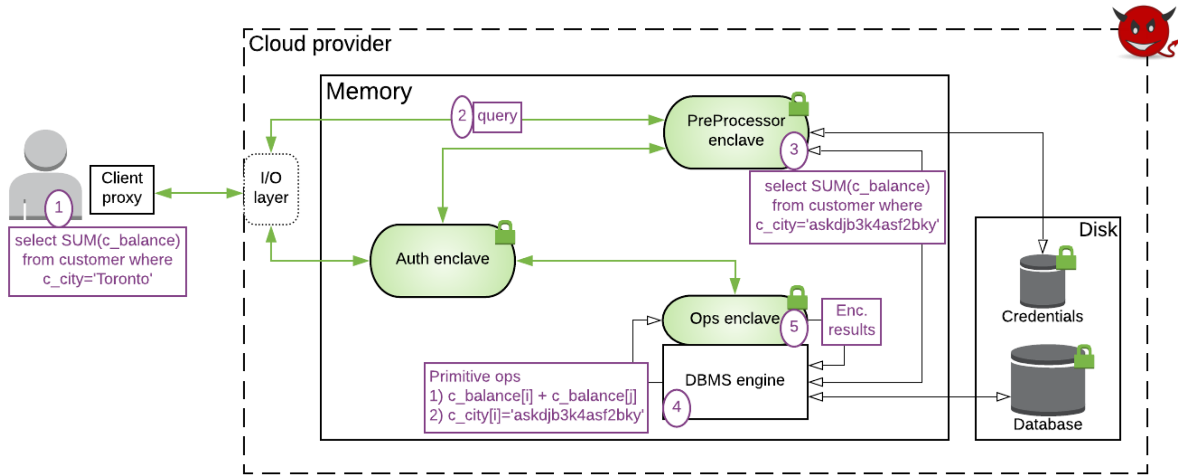


Figure 5.5: StealthDB architecture.

The life cycle of a query initiating from a client can be traced from steps 1 to 5. The lines with shaded arrows represent encrypted communication between those entities.

5.5 Architecture

The architecture of StealthDB is presented in Figure 5.5. As discussed in our third design, StealthDB makes extremely minimal changes to the underlying DBMS, with most of our components augmented on top of an unmodified DBMS. We will now go through the flow of database creation and query life-cycle, and explain each of our components in detail as needed.

5.5.1 Database creation

When a database is created, the database owner designs a database schema to define the structure of the database. During the schema creation, StealthDB allows the owner to identify the columns of the tables in the database which have sensitive information and use our *encrypted datatypes* for those columns. An encrypted datatype is used to represent values which are the encrypted versions of its corresponding plaintext datatype. For instance, encrypted integers are represented by the encrypted datatype *enc_int4* in

Figure 5.6. And, a database owner can issue the following command to create a table *item*

```
CREATE TYPE enc_int4 (
    INPUT          = enc_int4_in,
    OUTPUT         = enc_int4_out,
    INTERNALLENGTH = 45,
    ALIGNMENT      = int4,
    STORAGE        = PLAIN
);
```

Figure 5.6: Definition of *enc_int4*

with two columns of types encrypted integers and encrypted strings as in Figure 5.7.

```
CREATE TABLE item (
        id enc_int4 NOT NULL,
        name enc_text NOT NULL
);
```

Figure 5.7: Create table

StealthDB will encrypt the data values in an encrypted datatype using AES-GCM which is an authenticated encryption scheme providing both confidentiality and integrity of the data values. We will discuss about the key(s) used by this encryption during the DBMS initialization.

5.5.2 DBMS Initialization

When the DBMS is started, the following additional steps are performed for StealthDB.

Enclaves creation StealthDB creates three enclaves on the database server: the client authentication enclave Auth, the query pre-processing enclave PreProcessor and the operation enclave Ops. These enclaves are loaded by an *untrusted* DBMS runtime, but our system will later allow to *attest* that the correct code has been loaded into the enclaves. The clients use the remote attestation process and the publicly available measurements (hash) of the enclave code to ensure the correctness of the loaded programs in the enclaves. We will defer the explanation of this step and the functionality of these enclaves to the sections below.

To facilitate the communication between the users and the enclaves, StealthDB introduces an I/O layer on the server side. Its job is to simply redirect requests between the appropriate enclaves and the DBMS. This will also act as the *wrapper* program for the enclaves helping in processing their I/O requests and system calls. Note that this layer is untrusted and can be controlled by an adversary.

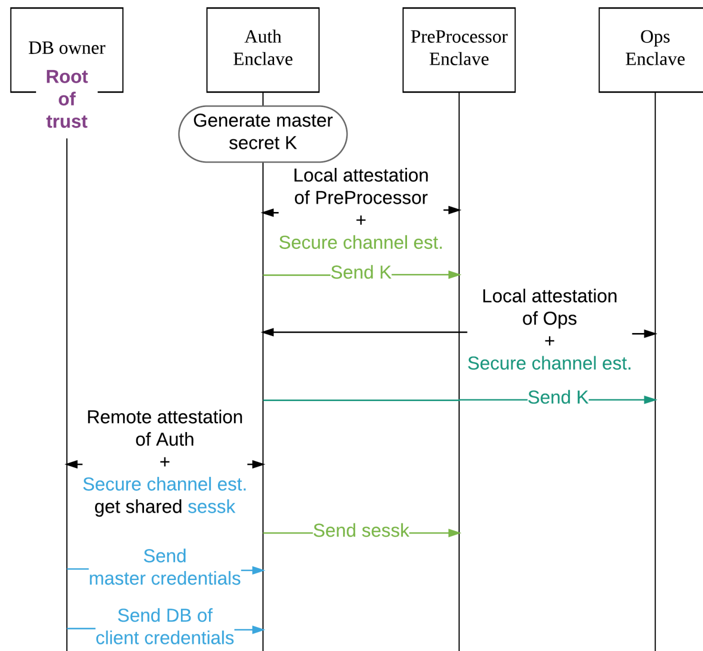


Figure 5.8: The authentication protocol of StealthDB

Key generation The initialization phase also involves generating a master secret key. StealthDB performs key generation inside the Auth enclave. Auth runs the KeyGen() function to sample a 128 bit secret key K at random for the AES encryption/decryption operations. In the current design, this master key K will be used to encrypt all the data values in the database. We do this for simplicity and our design can be extended to support an integration with a key management service to enable the usage of different keys for different clients or for different columns in the database.

Figure 5.8 outlines the key generation and transfer procedures. The master key K is then transferred to the PreProcessor and the Ops enclaves as follows. When the PreProcessor and Ops enclaves are created, they individually perform a local attestation with Auth and establish a secure channel with Auth. When the attestations succeed and after the secure channels are established, Auth's KeyTransfer() function uses the channels to send the master key K to PreProcessor and Ops. (On the other end, PreProcessor and Ops will run their KeyReceive() functions to complete these steps and receive K). On obtaining K, PreProcessor and Ops use SGX's sealing property to encrypt and store K for future use.

Transfer of credentials The final task of the initialization phase involves transferring the client credentials and access policies to Auth. A client (proxy) will authenticate to Auth. And, from the point of view of the DBMS, Auth (and PreProcessor) will act as a client who has complete access to the database. To facilitate this, the data owner first engages in a remote attestation protocol with Auth along with a secure channel establishment and if it succeeds, she sends the *master* credentials along with the database of client credentials and access policies to Auth through the established channel. On obtaining these, Auth uses the SGX seal operation to encrypt and store them.

5.5.3 Client authentication

One of the challenges we need to address is to make sure that only the authorized users can query the encrypted database system. For this, we design an authentication method built on top of an existing DBMS.

After the database server is started, it is now ready to accept connections from the clients. Here, StealthDB adds an authentication mechanism for the clients to authenticate to the Auth enclave. This works as follows.

First, the client proxy verifies that the DBMS has loaded the *correct* code into Auth, by performing the remote attestation (plus secure channel establishment) protocol with Auth as described in Section 5.2. Let sessk be the shared secret key obtained after its successful completion. The client will then authenticate to the Auth enclave using its credentials, say its password or its SSH key, through the established secure channel. On the server side, the I/O layer directs the client authentication requests to the CompleteClientAuth() function in Auth. CompleteClientAuth() unseals the client credentials database and uses it to verify the client credentials. If the client authentication completes successfully, the shared secret key sessk will be used as the *session key* for the client.

Once the client authentication is completed, the interaction with the client for query processing will be performed by the PreProcessor enclave. To facilitate this, the I/O layer will now invoke the TokenTransfer(ID, sessk) function in Auth to transfer the client "ID" and sessk to PreProcessor. This transfer will use the secure channel established between these enclaves during the master key transfer. The TokenReceive function of PreProcessor will seal and store sessk with ID as the additional authentication data during the seal operation.

5.5.4 Query execution

Now we will explain the working of query processing and execution in StealthDB for a client which has completed its authentication successfully. The design of StealthDB permits the use of an unmodified query driver (e.g. JDBC, ODBC, etc.).

When a client issues a query, the client proxy encrypts the entire query string using the session key sessk with its ID included in the additional authenticated data. On the server side, the I/O layer directs the client queries to PreProcessor. The QueryPreProcessing function first decrypts the query ciphertext using the session key sessk for ID. Then, it checks whether this client is permitted to run this query. Typically, a DBMS allows the DB owners to specify access control policies for the clients. In StealthDB, we rewrite the access control monitor inside PreProcessor. If the checks are passed, QueryPreProcessing identifies the data values in the query which correspond to the columns in the database using encrypted datatypes using our query parser, and AES-encrypts these data values using the master secret key K. The output of this step, encquery, is given to the DBMS for execution.

Note that the DBMS is oblivious to the changes made to the query. The *structure* of encquery is same as that of the query issued by the client. This lets the DBMS use an unmodified query parser to parse this query. But after the query is parsed and a query plan is obtained, we need to augment the DBMS with functions to operate on the encrypted datatypes. We do this as follows.

We first identify the set of *primitive* operators used by the underlying DBMS. Primitive operators are those further-indivisible operators used in query plans:

- *Comparators* such as $<, >, <=, >=, !=$, etc.
- *Math operators* such as $+, -, \%, *$, etc.
- *Hash functions* that are used to build some indexes.
- *Advanced math functions* such as *sin, cos, tan*, etc.

Traditionally, DBMSs define a functionality for each input datatype tuple supported by a primitive operator. StealthDB augments these with their functionalities when used with the corresponding encrypted datatypes as in Figure 5.9. Our implementation on Postgres implements primitive operator functionalities over the encrypted datatypes and include them as *extensions*.

```sql
CREATE OPERATOR = (
  LEFTARG = enc_int4,
  RIGHTARG = enc_int4,
  PROCEDURE = enc_int4_eq,
  COMMUTATOR = '=',
  NEGATOR = '<>',
  RESTRICT = eqsel,
  JOIN = eqjoinsel,
  HASHES, MERGES
);
```

Figure 5.9: Operator = for *enc_int4*. Here, *enc_int4_eq* will call the Ops enclave to decrypt the input, check their equality and output the result.

For every possible input datatype tuple, we define a function inside the Ops enclave. Suppose that we are given two encrypted data values (e_1, e_2) and an operator $\oplus$, the corresponding function inside Ops will perform:

1. *decryption* of the inputs e_1, e_2 using the master key to get plaintext values p_1, p_2,
2. *perform the operator function* to get $p_{\mathsf{output}} = p_1 \oplus p_2$,
3. *encrypt* the result p_{output} to get a ciphertext e_{output} using the master key (if specified by the design).

The number of inputs and outputs may of course vary depending on operator. Moreover, datatype conversions are also allowed in our model. For example, an encrypted integer may be converted to an encrypted string, and so on. Overall, we only perform a few basic operations (decrypt, primitive operator, encrypt) during the query execution inside the enclave.

Standard SGX *ocall/ecall* communication mechanism with enclaves is too slow when many calls are needed. To solve this, we implement an *exit-less* mechanism [174] for communicating with Ops. In [174], there is always one thread running inside an enclave listening for operator jobs. The DBMS uses our I/O layer to send jobs and receive replies via a communication queue. This method greatly improves performance by avoiding context switch for each call to the operator between trusted and untrusted zones, as we discussed earlier in Section 5.4.2.

There are also other inherent advantages with our design.

- When a client issues a query only involving unencrypted datatypes, the query processing and execution proceeds in the *native* way and hence with no overheads.

- A very interesting property is that our design also allows for computations between encrypted datatypes and unencrypted datatypes. The database owner here can also specify that the output of such computations should be encrypted to avoid leaking information about the encrypted inputs.

- Since our design implements only the primitive operators, it is easy for us to implement them inside Ops using data-oblivious methods [173] with a very small performance overhead to counter the side-channel attacks of SGX.

5.5.5 Encrypting indexes

The indexed columns, unlike the other columns in the table, need extra layers of protection. When the column is indexed into a B-tree, for example, the structure of the tree reveals the inequalities with respect to the values in the column even though the individual values in the tree are encrypted. The inequalities are available even to a snapshot adversary after index creation before any query is made to the database. We provide two modifications to reduce this leakage. First, we re-encrypt the individual values in the column when placing these encrypted values in an index structure. This unlinks the connection between the values in the table and the index. This unlinking is maintained for an adversary obtaining only a snapshot of the table and the indexes. Even for a slightly weaker persistent adversary which does not observe the system during the index creation, the inequalities observed from the index structure can be connected to the table values only when a query accesses the corresponding table row as part of its result. This change does not incur a performance overhead during the query execution in StealthDB .

The second change deals with this leakage on disk. StealthDB encrypts every page that is written to the files on disk corresponding to the indexes. We do this by encrypting the data right before it is written to the index files on disk, and decrypting the data read from the index files right after it is read from disk. In our implementation for Postgres, our changes to the codebase involve adding three lines of code to do this task. We create and run a fourth enclave Index_OP during the DBMS initialization which performs the encryption and decryption of the index data pages. And the three new lines are for retrieving the enclave ID, calling the encryption function inside Index_OP right before a FileWrite() of Postgres and for calling the decryption function inside Index_OP right after a FileRead(). The key used for these routines is generated and stored by Index_OP, and Auth attests the correct loading of Index_OP during the DBMS initialization.

5.5.6 Extensions

Encrypting logs Some of the log files reveal sensitive information about the queries even for a snapshot adversary on disk [119]. We can protect against an adversary accessing disk by encrypting the log files on disk in a way similar to our encryption of index files on disk. Perhaps, one could ask why we do not encrypt every page written to disk, not just indexes and logs. But the individual data items in the tables are already encrypted and we get no concrete security improvements by encrypting the individual disk pages containing those data items.

Key management In the current implementation, we use a single master key K to encrypt all the data values. K is sealed and stored on the disk by PreProcessor or Ops enclave when obtained from Auth. If and when the system is restarted, the enclaves are created again and a valid PreProcessor or Ops enclave can unseal the corresponding sealed components to obtain K. During this process, the AES-GCM encryption used in the SGX sealing provides confidentiality and integrity for the sealed component of K against any adversary. Also, when replicating the database across multiple machines, we can let the Auth in one of the machines to generate K and do a remote attestation to transfer it to the Auth enclaves in the other machines.

5.6 Security

The tradeoff between security, functionality, performance and the intrusiveness level decided by our design results in the leakage profile that we explain in this section.

First, we will discuss the effect of the SGX side-channel attacks on StealthDB . SGX is subject to various side-channel attacks as described in Section 5.2. The side-channel due to the application's page-level access pattern is a significant one and it is upto the application developer to design data-independent memory accesses for the application data to be secure. Our design addresses this side-channel by performing only primitive operations inside an enclave (Sections 5.4.2 and 5.5) and by using oblivious operators [173] for these primitive operations. We obviate the other software side-channels (except the cache-based ones) by simplifying the code inside the enclaves; running the primitive operations obliviously prevents these side-channels. The cache-based side channels [56, 77] though, are inherent to the x86 architecture and requires patching from Intel. (Also, these are instances of active attacks, which in general StealthDB does not protect against).

Now, let us discuss the leakage profile of StealthDB . As mentioned in our threat model in Section 5.3.2, StealthDB protects against semi-honest or passive adversaries. It does not provide integrity guarantees to the clients on the correctness of the query results. Neither does it provide confidentiality guarantees against an actively malicious adversary with side-information on the plaintext values encrypted in DB. We will first detail the leakage profile of StealthDB for different variants of semi-honest adversaries and through a series of security claims we will argue that StealthDB does not leak any more information than what is part of the leakage profile. Our evaluation is with respect to the architecture we propose, and hence independent on the specific underlying DBMS engine.

5.6.1 Leakage profile

StealthDB encrypts the individual data items, rather than an entire column or table at once, and hence this mandates a thorough leakage profiling. We classify the admissible adversaries as in [90] and quantify leakage profiles during the high level operations, Init and Query, of a DBMS for those adversaries. Init involves loading the database in the untrusted server to be ready for querying, and Query involves the client querying the database to get the required results. Note that a query in StealthDB can involve any operator supported by the underlying DBMS (Eg. relational, arithmetic and logical operators for a transactional DBMS).

We analyze the security of StealthDB against passive or semi-honest adversaries. We further classify the adversaries into snapshot and persistent adversaries. A snapshot adversary gets a snapshot to the memory of the system whereas a persistent adversary observes the memory of the system throughout its execution. We motivate these adversarial types in Section 5.3.2.

Let DB denote the database that we try to securely operate. DB includes all the data structures used by a database (for eg., tables, indexes, views, foreign tables) along with their contents. We will now define the leakage entities formally and prove the security of StealthDB through the simulation paradigm. To prove the security though a simulation paradigm, we first define the set of admissible adversaries. We then construct a polynomial-time simulator Sim which, with only the leakage entities as input, produces a view ("ideal view") that is indistinguishable for any admissible adversary from the view ("real view") obtained through a real execution of the StealthDB system. This will prove that the upper bound on the information leaked by StealthDB to the adversary are the inputs used by the simulator to produce the ideal view. Figure 5.10 provides the formal simulation security definition for an encrypted database system using trusted hardware definition.

$$\begin{array}{ll}
\mathsf{Real}_{\mathsf{EncDB}}(1^\lambda): & \mathsf{Ideal}_{\mathsf{EncDB}}(1^\lambda): \\
(K, \mathrm{EDB}) \leftarrow \mathsf{Init}(1^\lambda, \mathrm{DB}) & \mathrm{EDB} \leftarrow \mathsf{Sim}^{\mathcal{L}}(1^\lambda) \\
\mathrm{encres} \leftarrow \mathsf{Query}^{\mathsf{HW}(\cdot)}(\mathrm{EDB}, \mathsf{encquery}) & \mathrm{encres} \leftarrow \mathsf{Query}^{\mathsf{Sim}^{\mathcal{L}(\cdot)}(\cdot)}(\mathrm{EDB}, \mathsf{encquery})
\end{array}$$

Figure 5.10: Security definition for an encrypted database system using trusted hardware. An EncDB construction is secure if, for all admissible adversaries, there exists an efficient Sim such that:

$$|\Pr[\mathsf{Adv}(\mathsf{Real}_{\mathsf{EncDB}}) \to 1] - \Pr[\mathsf{Adv}(\mathsf{Ideal}_{\mathsf{EncDB}}) = 1]| < \mathrm{negl}(\lambda)$$

where $\mathsf{Adv} = (\mathsf{Adv}_1, \mathsf{Adv}_2)$. Adv_1 runs the Real or the Ideal experiment, whereas Adv_2 obtains information about the experiment from Adv_1 depending on the adversarial type being studied and produces the output 0 or 1. A snapshot Adv_2 obtains a snapshot of the system, when desired, from Adv_1, whereas a persistent Adv_2 completely observes the EncDB system while Adv_1 is running the experiment. Adv_1 is tasked with just running the EncDB system; a semi-honest Adv_1 will run as per the specifications, and an actively malicious Adv_1 will run the system as desired to maximize the information obtained by Adv_2. The access to HW is treated as an oracle as in Iron and Sim simulates the oracle in the Ideal experiment. The HW oracle provides interfaces to the enclaves used (in StealthDB , they are $\mathsf{Auth}(), \mathsf{PreProcessor}(\mathsf{encquery})$ and $\mathsf{Ops}(\{\mathsf{input}\}, \mathsf{op})$). When Query is invoked on a query, Sim will obtain the leakage $\mathcal{Q}$ corresponding to a query from $\mathcal{L}$.

This definition is inspired by the HW definition of Iron who define simulation security for functional encryption using trusted hardware HW.

The leakage entities of interest to StealthDB are as follows:

- Let $\mathcal{S}_t$ indicate the *shape* of the database at time[3] $t \geq 0$ which includes
  - the database schema,
  - the shape of the tables and (database) views i.e., the number of rows and columns in the tables and views,
  - the shape of the indexes (for eg. the shape of a B-tree index reveals the number of keys in each internal node of the tree).

[3] "Time" t refers to the epoch at which the data-structure is observed or collected from the system

More importantly, $\mathcal{S}_t$ does not include the contents of any of the data structures in the database. This entity varies with time depending on the queries run on DB.

- Let $\mathcal{Q}$ denote the leakage associated with a query execution. In StealthDB , $\mathcal{Q}$ is upper bounded by the union of the plaintext outputs of the Ops enclave invocations.
- Let $\mathcal{M}_t$ denote the leakage associated with the logs and the miscellaneous data structures maintained by a DBMS at time t to aid in its operations (including various profiling activities and recovery from unexpected failures).

In StealthDB , the entities $\mathcal{Q}$ and $\mathcal{M}$ are dependent on the underlying DBMS that StealthDB builds on. In Section 5.7, we discuss the information that can be inferred from $\mathcal{S}$, $\mathcal{Q}$ and $\mathcal{M}$ for some real-world data structures and queries.

Note that $\mathcal{S}$, $\mathcal{Q}$ and $\mathcal{M}$ are leakages with respect to DB. We now define the leakage entity $\amalg$ with respect to a query. In StealthDB , before the query is executed (after output by PreProcessor), the query structure is revealed but not the constants in the query which are encrypted with the semantically secure encryption. With $\mathcal{Q}$ being the leakage during the execution of this query, the total leakage of a client query to the server is upper-bounded by the union of $\amalg$ and $\mathcal{Q}$ for this query.

- Let $\amalg$ indicate the leakage about the query before the DBMS begins processing it.

Typically, $\amalg$ will be a subset of the DB based leakages. In a real-world DBMS, $\amalg$ might just be a subset of $\{\mathcal{M}_t\}$ since the details about input queries are usually logged and checkpointed.

We will now prove the leakage profile of StealthDB during different phases of its execution. All the following claims rely on the fact that no information (other than its length) about the key K used to encrypt the data is revealed to an adversary (Claim 5.6.4). We would rely on the following security properties:

1. Remote and local attestation provided by SGX are secure according to Section 5.2.
2. The confidentiality of the intermediate values of the computation and the integrity of the computation from SGX.
3. The confidentiality and integrity provided by the authenticated encryption of AES-GCM.
4. The confidentiality of ElGamal encryption - used during secure channel establishments.

Init **phase** StealthDB only leaks the initial shape $\mathcal{S}_0$ during the Init phase. This is better than the OPE or ORE based designs [182, 175] which leak the '¡' relation between all the values in the OPE/ORE encrypted columns.

Claim 5.6.1. *After the completion of* Init *and before any call to* Query *is made, StealthDB leaks at most $\mathcal{S}_0$.*

Proof. The proof is straight-forward and the high-level idea is as follows. Sim obtains $\mathcal{S}_0$ from the leakage oracle $\mathcal{L}$ and outputs encryption of zeros according the shape $\mathcal{S}$ as EDB. An adversary Adv_2 that distinguishes the simulated EDB from a real EDB will break the semantic security of the encryption scheme. □

Query phase We will first prove the leakage of StealthDB for adversaries which obtain snapshot access to the memory. A snapshot adversary in StealthDB learns at most the shape $\mathcal{S}$ and the leakage $\mathcal{M}$ due to the miscellaneous information maintained at the time of the snapshot. $\mathcal{M}$ is further upper-bounded by the union of $\mathcal{Q}$ from the queries executed recently. More formally, we have the following claim. The proofs in this section are in the Appendix.

Claim 5.6.2. *Consider a polynomial-time snapshot adversary on StealthDB obtaining the snapshot at time t. Let $t' \leq t$ be the latest time epoch before t for which the logs and miscellaneous data structures remain in memory and not written to disk. The adversary learns at most $\mathcal{S}_{t'}$ of the* DB *being operated and $\mathcal{Q}$ of the queries executed between t' and t. If the log items are encrypted in memory, the adversary learns at most $\mathcal{S}_t$.*

Proof. Adv_2 would query the snapshot of the system at time t. Sim sets up EDB as encryption of zeros of arbitrary shape $\mathcal{S}_0$ and answers the Ops queries arbitrarily till t'. At time t', Sim obtains $\mathcal{S}_{t'}$ from $\mathcal{L}$ and rewrites EDB with encryption of zeros according to $\mathcal{S}_{t'}$. For each query run between t' and t, Sim obtains $\mathcal{Q}$ from the oracle $\mathcal{L}$ and answers the Ops queries accordingly. This way, the execution of the Real and Ideal experiments and the corresponding shapes of EDB are consistent at time t assuming a deterministic order of execution for EDB.

We will now argue that the Real and the Ideal experiments are indistinguishable. When Adv_2 obtains the snapshot of the system at time t, it obtains EDB along with the logs and miscellaneous data structures maintained at time t. Given that the shape of EDB is consistent between the two experiments at time t, semantic security ensures that a real EDB is indistinguishable from the encryption of zeros. Logs, etc. for queries before time

t' are encrypted and written to disk. Hence, they do not reveal any information about the data items in DB. The logs maintained in between t' and t are also consistent between the two experiments and are consistent.

If the logs and the other data structures are encrypted in memory, Sim can behave arbitrarily till t and just rewrite EDB according to $\mathcal{S}_t$ at time t. Assuming the size of logs to not reveal sensitive information, the Real and the Ideal experiments are indistinguishable to Adv_2. □

We will now prove the leakage for a persistent adversary. A persistent adversary in StealthDB learns the plaintext outputs of the Ops enclave invocations throughout its observance. More formally, we prove the following claim.

Claim 5.6.3. *A polynomial-time semi-honest adversary that has persistent access to the memory during the StealthDB execution on a* DB *learns at most the shape* $\{\mathcal{S}_t\}_{t\geq 0}$ *of* DB *and the query-execution associated leakage* $\mathcal{Q}$ *for all the queries executed, where* $\mathcal{Q}$ *is the union of the plaintext outputs of* Ops *invocations during the execution of the query.*

Note that this theorem implies that the miscellaneous data structures $\mathcal{M}$ maintained or the parts of DB accessed during query execution do not leak more information than $\{\mathcal{S}_t\}$ and $\{\mathcal{Q}\}$ to a persistent adversary.

Proof. We again give the high-level idea of this simple proof here. During Init, Sim obtains the shape $\mathcal{S}$ from the leakage oracle $\mathcal{L}$ and encrypts zeros as EDB according to $\mathcal{S}$. This EDB is indistinguishable from a real EDB by the semantic security of the encryption scheme. Further, during the execution of Query, the values in DB are only used inside the Ops enclave. With a deterministic execution of EDB, Sim uses $\mathcal{Q}$ obtained from $\mathcal{L}$ to answer the plaintext outputs. For the encrypted outputs, Sim produces encryption of zeros as Ops output and this is again indistinguishable from the encryption of the real values by the semantic security of the encryption scheme. □

5.6.2 Security of K during StealthDB execution

Outline. We will argue here that no information about the master key K is revealed; also that only the *permitted* clients can make the DBMS execute queries. This will be a precursor to the leakage profile analysis above.

Claim 5.6.4. *The confidentiality and integrity of the master key* K *is ensured throughout the StealthDB execution.*

Proof. The database owner forms the *root of trust* as in Figure 5.8. The owner is involved in a remote attestation protocol with Auth to check the correctness of the code and the constants loaded into Auth against the publicly available *expected measurement* of Auth. (The constants loaded into Auth include the expected measurements of PreProcessor and Ops). The master credentials for the database is transferred to a valid Auth. And, the security of SGX remote attestation guarantees the validity of Auth. From this point, the trust is transferred to Auth. Auth generates the master key K.

The master K is then transferred to the other enclaves PreProcessor and Ops by Auth through the secure channels established on top of local attestation. The security of local attestation ensures that Auth establishes secure channels with only those PreProcessor and Ops whose measurements match the *expected* hardcoded ones. Hence, K is transferred only to the correct instances of PreProcessor and Ops. Here, the confidentiality provided by the public key cryptography used in the secure channel establishment (on top of the authenticity from attestation) and the confidentiality and integrity of AES-GCM ensure that no information about K except its length is leaked to an adversary during the transfers.

Now, there are only two more operations which involve K. First, when K is used to AES encrypt and decrypt data values, the SGX security guarantees combined with the obliviousness of the AES-NI instructions ensure that no intermediate values about K are leaked. Finally, K is also sealed and stored on the disk for later retrieval. Here, the SGX sealing process provides confidentiality and integrity to K. □

Claim 5.6.5. *During the query execution phase, a query which reaches the DBMS for execution satisfies the access control policies for the client requesting the query.*

Proof. The security of remote attestation also ensures that the database owner transfers the client credentials database only to a valid Auth. When a client proxy initiates a connection with the DBMS, a valid Auth establishes a session with the client only if the client has valid credentials. Next, Auth transfers the session key sessk (shared with the client) only to a valid PreProcessor. This is ensured by the security of local attestation. Now, when the client issues a query, the I/O layer relays it to PreProcessor and PreProcessor parses the query and proceeds only if the query satisfies the access policies of this client. Since there is no other interface for the client to issue a query to the semi-honest DBMS, StealthDB ensures that the semi-honest DBMS only executes a query from a valid client satisfying the access policies provided by the database owner. □

5.7 Concrete leakage profiles

The formal discussion above provided an upper bound on the leakage in terms of abstract leakage entities. The definition of the shape $\mathcal{S}$ is concrete from the definition. But, $\mathcal{Q}$ and $\mathcal{M}$ depend on the underlying DBMS that StealthDB builds on. $\mathcal{M}$ is typically We will now concretize this for the different operations performed on encrypted data.

- *Arithmetic operations*: Some examples of arithmetic operators include +, -, %, * and advanced ones like sin, cos, log. For these operators, we provide the same security as a fully-homomorphic encryption (FHE) on the computation performed on individually encrypted data items. As in FHE, StealthDB does not reveal any information to a semi-honest adversary about the intermediate values of an arithmetic computation involving encrypted inputs and outputs, other than their length (as multiples of 128 for AES). Consider a simple example query from a TPC-C transaction: `update table_warehouse set w_ytd = w_ytd + constant where w_id = constant2`. StealthDB reveals no information about the values in the column w_ytd during the execution of this query.

- *String operations*: String operations like substring and wildcards have no leakage, other than the length of inputs and outputs (upto a multiple of 128), with them being encrypted.

- *Relational operations*: A real-world DBMS uses indexes to perform the relational operations like comparisons and joins efficiently. The $\mathcal{Q}$ for a query using an index, say a B-tree, includes the comparison results of the parts of the B-tree explored by the query. As the values in the index are re-encrypted versions of the values in the table, the comparison results are useful only when the corresponding values are accessed in the table. When a row becomes part of query results, an adversary usually can link it to the corresponding value in the index. From this, it can use the Ops output history to obtain the comparison results between the indexed value in this row with the indexed values from the other accessed rows Hence, the information revealed by $\mathcal{Q}$ in StealthDB is the comparison results for indexed values in the rows accessed by the queries. In the worst case, our leakage against persistent adversaries reduces to ORE for the parts of the indexes explored by the queries.

  There is also a non-trivial information leakage to a persistent adversary that only has access disk, and not memory. The index pages on disk that are modified during checkpointing reveal some inequalities within the data being inserted or modified. In Postgres, for instance, the index file stores data as 8 KB pages. When a new value is

inserted into the table, only the pages that need to be changed are marked as dirty in the memory and eventually changed on disk.

For any other DBMS, the precise information revealed by $\mathcal{Q}$ varies based on its query execution and log maintenance procedures.

5.8 Implementation and Performance

5.8.1 Implementation details

We implement StealthDB in C and C++ on top of Postgres 9.6 as an extension that loads new SQL functions, encrypted data types and operators and index support methods for the encrypted datatypes. The command *CREATE EXTENSION stealthdb* loads the files *stealthdb.so* (the main library), *enclave_stealthdb.so* (part of the code which is executed in enclaves), *stealthdb.control* (the version control file), *stealthdb.sql* (definitions of new defined functions) into the system. For instance, the function *enc_int4_cmp* in Figure 5.11 compares two *enc_int4* values and returns {-1, 0, 1}. The function *enc_int4_cmp* in Figure 5.12 is

```
CREATE OR REPLACE FUNCTION enc_int4_cmp(enc_int4, enc_int4)
RETURNS integer
AS '$libdir/encdb'
LANGUAGE C IMMUTABLE STRICT;
```

Figure 5.11: Example of a new function definition in stealthdb.sql

```
PG_FUNCTION_INFO_V1(enc_int4_cmp);
Datum
enc_int4_cmp(PG_FUNCTION_ARGS)
{
    char *c1 = PG_GETARG_CSTRING(0);
    char *c2 = PG_GETARG_CSTRING(1);
    int resp = ENCLAVE_IS_NOT_RUNNING, ans = 0;
    resp = enc_int32_cmp(c1, c2, &ans);
    sgxErrorHandler(resp);
    PG_RETURN_INT32(ans);
}
```

Figure 5.12: Example of new defined function implementation in stealthdb.c

executed in an enclave. We implement our query *pre-parser* in the PreProcessor enclave on the server side to encrypt the data values in queries and this design helps in avoiding changes to the client JDBC or ODBC drivers of the system. Our approach can be extended to other SQL-like database using user-defined functions. Though database systems like MySQL do not allow creating independent extensions like Postgres to include our changes,

these changes are not intrusive and completely independent of the improvements to the core database operations. To protect against the side-channel attacks on SGX, we make every operation inside an enclave oblivious by leveraging AES-NI and CMOV instructions. The source code of Postgres 9.6 has about 700K lines of code while StealthDB has about 5k lines of code with 1.5k lines run in enclaves.

5.8.2 Performance evaluation

To measure StealthDB's performance, we use an Intel Xeon E3 3.60 GHz server with 8 cores and 16 GB of RAM. In our experiments, we measure the throughput and latency of StealthDB using the TPC-C trace and compare the results with an unmodified Postgres 9.6 which works with unencrypted data. TPC-C [1] simulates the activity of a wholesale supplier, which includes database transactions for entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. This is a benchmark for measuring the performance of the on-line transactional processing (OLTP) systems. The results in our evaluation were obtained by averaging multiple 1000 second runs with check-pointing turned off. We ran our experiments with the number of clients varying from 1 to 10 and with a single-threaded enclave used by all the client connections. The number of clients can be further increased if a multi-threaded enclave is used. Our first set of experiments leave the IDs in the TPC-C tables (e.g. w_id, o_w_id, etc.) unencrypted. The tested database includes nine tables with about 10 million rows in total. This is about 2GB of unecrypted data and when encrypted for StealthDB gives an encrypted database of size 7GB.

Throughput Figure 5.13 shows the throughput for the TPC-C benchmarking for different scale factors. StealthDB incurs an 4.7% overhead over the unmodified Postgres for the scale factor $W = 1$ and around 30% overhead for $W = 16$. This is sufficient for many real-world transactional systems for the security advantages.

Latency We measure the end to end TPC-C transaction latency for StealthDB with the scale factor $W = 16$. This includes the time for our query pre-parser.

Table 5.1 and Figure 5.14 compare the median and average latency for StealthDB with the unmodified Postgres. The 90th percentile of the latency of StealthDB system is 7.2 milliseconds which results in a 22% overhead over the unmodified version.

We also test the performance of StealthDB when the IDs are encrypted with AES-GCM. That results in about 40% storage overhead, 3x throughput decrease over StealthDB with

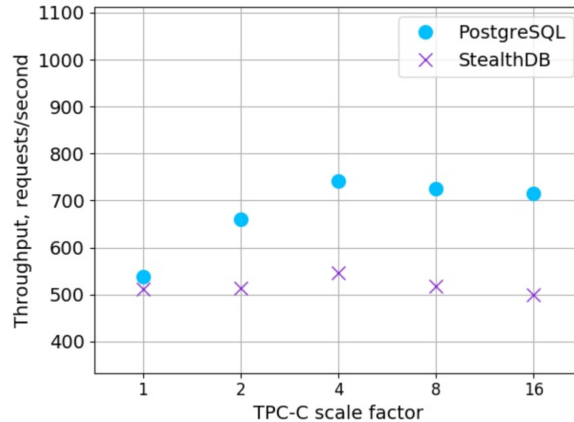


Figure 5.13: TPC-C benchmarking throughput for running under Postgres and StealthDB with different scale factors

	Median	90th percentile
PostgreSQL	1.6	5.9
StealthDB	2.8	7.2

Table 5.1: Latency statistics of TPC-C requests, ms

unencrypted IDs. And the latency is 3.6 times of that of the version with unencrypted IDs. The IDs in the TPC-C tables are just counters, hence encrypting them do not offer any concrete security advantages.

5.9 Related Work

This section builds on the comparisons from the introduction. The work most similar to ours is Cipherbase [20]. But the trusted on-premise key loading phase for every FPGA device, and cloud operator controlled "shell" monitor [17] inside an FPGA make FPGAs unsuitable for being used as a *trusted hardware* in the cloud. In terms of performance, [20] achieves about 10% better throughput than ours, but they skip two TPC-C transactions in their evaluation. Our evaluation with the complete TPC-C benchmark finds that these two transactions have the highest latency overheads. Similar bottlenecks are expected for Cipherbase with FPGAs. And, as expected, we achieve much lower latency (4×) over

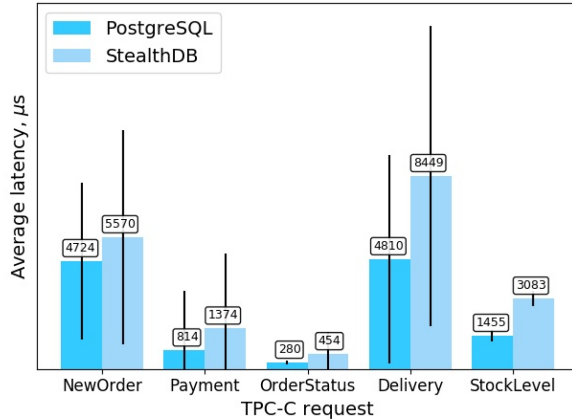


Figure 5.14: Average latency and standard deviation for TPC-C requests under Postgres and StealthDB.

the FPGA implementation. TrustedDB [26] uses the IBM secure co-processor to perform operations, but with large portions of the DBMS engine executed inside the trusted zone. The IBM co-processor incurs high overheads for transactional workloads and also, this design is not suitable for SGX for both security and performance reasons as we discussed in Section 5.4.2.

CryptDB [182] uses a hybrid of encryption schemes to support subset of SQL functionality. Their underlying large leakage profiles often result in data compromise [169, 118]. Performance-wise, [182] achieves a similar throughput decrease as ours, but only when evaluated with the individual queries from the TPC-C transactions over a 20× smaller dataset. Arx replaces OPE scheme with a special garbled-circuit based searching method [181]. Garbled circuits however introduce large computational and storage overheads.

A few works studied how to build versions of encrypted databases with SGX. VC3 system proposes an architecture for analytical MapReduce jobs in cloud settings [190]. Opaque studies how to leverage SGX to secure distributed analytical workloads in Spark systems [211]. A concurrent work of ours, ObliDB [85], obtains an oblivious database supporting both transactional and analytical workloads. But, their solution involves extensive changes to the underlying DBMS engine, and does not scale well for transactional workloads. Another concurrent work, EnclaveDB [184], provides strong security guarantees against persistent and active adversaries. However, this is achieved by placing larger components of DBMS inside enclaves assuming the existence of large enclaves, in the

order of GBs, which is much greater than the 128MB available today.[4] They also ignore the access pattern and other side-channel attacks. In summary, [184] focuses on a different design space assuming how future trusted hardware designs may look., while our work focuses on building encrypted database from standard trusted hardware available today.

HardIDX [89] investigates how to perform range queries obliviously over B+ tree indexes inside an enclave, leaking only the parts of the database accessed per query. But, they only consider a static database, and the client should generate the full B+ tree index locally and store it in the server only for the querying. We can incorporate their ideas in StealthDB if we were to only support static databases and powerful clients. Also, [89] just prototypes index searches, whereas we architecture and build a complete encrypted database system.

A number of works study how to load unmodified applications into enclaves [32, 21, 202, 127]. These approaches work well for applications that process small data sizes, but do not scale well to larger workloads due to SGX limitations. Also, increasing the complexity of the codebase inside the enclaves aggravates the security risks associated with SGX [145].

OSPIR-OXT [63, 62, 86], SisoSPIR [133] and BLIND SEER [177] build encrypted database systems from scratch with provable security guarantees for a subset of functionality based on different cryptography tools. There are also multitude of other works which provide improvements over security or specific functionalities of a database, but they are not implemented or integrable with a mature DBMS. A recent systematization work by Fuller et al. [90] provides are great summary of the state-of-art research in encrypted database systems. Fully homomorphic encryption [95] is another powerful cryptographic primitive which enables an untrusted user to perform arbitrary computations on encrypted data without learning any information about the underlying data. But the current constructs for doing this are very far from being practical [124]. In general, while theoretical security of systems built based on cryptographic methods can be high, the *real-world* security of the system relies on the multitude of factors: correct implementations of non-trivial crypto algorithms, meta-data contents, information in log files, etc. Hence, it is not possible to argue their security just from the security of the crypto protocols used.

5.10 Conclusion

StealthDB offers a scalable encrypted cloud database system with full SQL query support with a modest 30% throughput decrease and $\approx$ 1 ms latency increase while providing

[4] It is an open question to achieve larger enclaves efficiently while providing security against physical attacks. SGX enclaves use Merkle-trees for integrity which adds logarithmic overhead to every access.

reasonable end-to-end security guarantees. StealthDB can be implemented in any newer generation Intel CPUs.

Supporting analytical workloads, reducing the leakage profile and protecting against active adversaries (i.e., providing integrity to the system) while maintaining our design principles are interesting open questions in this space. The source code of our implementation is also open-sourced and available at https://github.com/cryptograph/stealthdb.

Chapter 6

Quantum resistance of SGX

6.1 Introduction

This chapter reviews the existing work on the quantum resistance of the cryptography used in SGX. The security of SGX, as we discussed in previous chapters, relies on a multitude of factors. There has been a significant research on SGX's hardware access controls, key derivation and management and its micro-architectural details. This is because these have been the current avenues of attack and need imminent improvement for SGX to be deployed in the proposed security applications. But in the long-term with the advent of quantum computers, there is also a need to upgrade the cryptography used by SGX or any trusted hardware design [167]. Motivated by the advances in quantum computing research, there is a widespread effort including the standards organizations [171, 172, 24] to develop and transition to quantum-resistant cryptographic algorithms. The standardization efforts right now are focused on quantum-resistant crypto algorithms for key-exchange and digital signatures, but there has been extensive research from academia and industry on designing candidates for quantum-resistant versions of every cryptographic primitive. In this chapter, we first review the cryptography used by SGX and assess its security against quantum adversaries. We then survey the (surprisingly limited) research in the direction of designing these core cryptographic primitives based on quantum-resistant cryptographic assumptions.

6.2 Cryptography used by SGX core

In this section, we will discuss the cryptography used by the Intel facilities and the SGX hardware to provide the security properties of SGX. The following cryptographic primitives are used:

1. an Enhanced Privacy ID (EPID) scheme [54].
2. AES in a "tweaked" CTR mode. The tweak is in the counter which includes components related to the address of the memory block and time [120].
3. Carter-Wegman MAC algorithm [206, 120].

6.3 Enhanced Privacy ID (EPID)

The EPID scheme forms the core of the remote attestation process of SGX. Remember that remote attestation enables a platform running SGX to prove the following two aspects to a remote user:

- The user program is running on a genuine processor under the SGX restrictions.
- The correctness of the program run inside an SGX enclave, that it is the same program as provided by the user.

EPID helps in achieving these properties. At a high level, an EPID construction can be viewed as a group signature scheme [68, 33] with Intel being the group manager issuing signing credentials to every machine that has SGX support.[1]

Relation to other cryptographic primitives But EPID is different from group signatures in that the group manager cannot trace or open a signature to identify the group member who generated that signature. On the other hand, EPID supports revocation of group members to compensate for the lack of traceability. Intel does not want to identify

[1] A secret (root provisioning key) is hardcoded in the e-fuses of the SGX machines during manufacturing which let them prove to Intel that the machine has SGX support. But once the machine has proved to Intel and obtained the signing credentials, the machine can sign enclave measurements while being anonymous to Intel. There are also ways to modify this to identify signatures (attestations) produced by the same machine [57].

the machines based on their attestation signatures, but it is necessary to revoke the signing capabilities of the machines which got compromised. As we will notice from the definition, EPID is more close to the primitive of Direct Anonymous Attestation (DAA) [53]. In fact, EPID is a DAA with additional revocation capabilities compared to a DAA. The primitive of Blacklistable Anonymous Credentials (BLAC) [203] also follows the same definition of EPID. But, we will use the term EPID throughout this thesis.[2]

In the rest of this section on EPID, we will first provide its definition and a generic construction from one-way functions inspired by the one for group signatures proposed by [33]. We will then discuss the quantum-resistance of the EPID scheme used in SGX and the research on EPID and EPID-like schemes based on quantum-resistant assumptions.

6.3.1 Definition

An EPID scheme involves four types of users: an Issuer $\mathcal{I}$, a revocation manager $\mathcal{R}$, platforms $\mathcal{P}$ and verifiers $\mathcal{V}$. A revocation manager $\mathcal{R}$ maintains two revocation lists: a secret key based revocation list $\mathsf{Key\text{-}RL}$, and a signature based revocation list $\mathsf{Sig\text{-}RL}$. A signing key used by a platform is considered revoked either if it is in $\mathsf{Key\text{-}RL}$ or if it is used to produce any of the signatures in $\mathsf{Sig\text{-}RL}$. Typically, a compromised key is added to $\mathsf{Key\text{-}RL}$ and a signature believed to be from a compromised platform by $\mathcal{R}$ is added to $\mathsf{Sig\text{-}RL}$.

An EPID scheme is made up of the following algorithms.

$\mathsf{Setup}(1^\lambda) \rightarrow (\mathsf{gpk}, \mathsf{gsk})$ The setup algorithm run by the issuer $\mathcal{I}$ takes as input a security parameter λ and outputs a group public key gpk and a group secret key gsk.

$\mathsf{Join}\langle(\mathsf{gpk}, \mathsf{gsk}), \mathsf{gpk}\rangle \rightarrow \langle\mathsf{cert}_i, (\mathsf{sk}_i, \mathsf{cert}_i)\rangle$ $\mathcal{I}$ and $\mathcal{P}_i$ run the (possibly interactive) join protocol with $(\mathsf{gpk}, \mathsf{gsk})$ and gpk as their respective inputs for $\mathcal{P}_i$ to join the group. At the end of the protocol, $\mathcal{P}_i$ obtains its signing key sk_i and a certificate showing that it is part of the group, and $\mathcal{I}$ obtains only the certificate.[3]

$\mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL}) \rightarrow \sigma/\perp$ The signing algorithm run by a platform $\mathcal{P}_i$ takes in a group public key gpk, its signing key sk_i, a message m to sign and a list of revoked

[2] EPID was designed by researchers from Intel and was a follow-up or a concurrent work to BLAC [203]. But due to the heavy use of the term EPID in the SGX literature, we will stick with the term EPID for this primitive.

[3] [54] defined the Join protocol such that $\mathcal{I}$ does not get any output. But some non-secret components of sk_i will be available to $\mathcal{I}$ at the end of the protocol. This is not a concern until anonymity and the other security properties of EPID are satisfied by the protocol.

signatures $\mathsf{Sig\text{-}RL}$. The output is a signature σ if sk_i is not used to produce any signatures in $\mathsf{Sig\text{-}RL}$, else a $\perp$.

$\mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Key\text{-}RL}, \mathsf{Sig\text{-}RL}, \sigma) \rightarrow 0/1$ The signature verification algorithm takes in a group public key gpk, a message m, revocation lists $\mathsf{Key\text{-}RL}$ and $\mathsf{Sig\text{-}RL}$ and a signature σ. It outputs 1 to accept σ and 0 to reject it.

$\mathsf{Revokesk}(\mathsf{gpk}, \mathsf{Key\text{-}RL}, \mathsf{sk}_i) \rightarrow \mathsf{Key\text{-}RL}$ The secret key revocation algorithm updates the key revocation list $\mathsf{Key\text{-}RL}$ to include the input sk_i.

$\mathsf{Revokesig}(\mathsf{gpk}, \mathsf{Sig\text{-}RL}, \sigma, m) \rightarrow \mathsf{Sig\text{-}RL}$ The signature revocation algorithm updates the signature revocation list $\mathsf{Sig\text{-}RL}$ to include the input signature, message pair (σ, m).

Here, for the clarity of the definition we let a platform $\mathcal{P}_i$ obtain at most one secret key sk_i.

The algorithms of an EPID scheme satisfy the following three properties.

Correctness The correctness of an EPID scheme guarantees that every signature generated by a platform will be accepted by the verifier unless the platform is revoked. The formal definition is as follows. Let Σ_i be the set of signatures generated by $\mathcal{P}_i$. We have that

$$(sk_i \notin \mathsf{Key\text{-}RL}) \wedge (\Sigma_i \cap \mathsf{Sig\text{-}RL} = \emptyset)$$
$$\Rightarrow \mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Key\text{-}RL}, \mathsf{Sig\text{-}RL}, \mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL})) = 1$$

Anonymity An EPID scheme provides anonymity to signatures produced by the platforms even from the group manager. At a high level, the anonymity game involves an adversary trying to distinguish between the signatures produced by two platforms on a challenge message m^* and $\mathsf{Sig\text{-}RL}^*$ of its choice. The adversary can run Setup before providing the challenge $(m^*, \mathsf{Sig\text{-}RL}^*)$. Also, before and after the challenge phase, adversary can run Join with the platforms of its choice, query signatures for arbitrary m and $\mathsf{Sig\text{-}RL}$ from $\mathcal{P}_i$ of its choice and revoke $\mathcal{P}_i$s of its choice. We even let this adversary act maliciously by not following the specifications of the protocol when running its algorithms. Although, in all these steps, we have restrictions that avoid providing trivial advantages to the adversary in the security game. A secure EPID scheme prevents any polynomial time adversary from gaining non-negligible advantage (over guessing) in this security game. We direct to [54, 40] for the formal definition as we will not use it here.

Unforgeability EPID schemes provide an unforgeability guarantee for the signatures when all the platforms that have been corrupted during the unforgeability game are revoked using either Revokesk or Revokesig. A strong-unforgeability (and not just existential unforgeability) like guarantee is provided in EPID by also allowing the adversary to produce a signature σ^* on a particular $(m^*, \mathsf{Sig\text{-}RL}^*)$ as a valid forgery irrespective of it querying Sign on the same $(m^*, \mathsf{Sig\text{-}RL}^*)$ for some platform, as long as σ^* was not an output of one of those queries on $(m^*, \mathsf{Sig\text{-}RL}^*)$. A formal unforgeability game is available in [54, 40].

Quantum resistance All the available versions of SGX use the EPID scheme in [54]. It is based on the q-Strong Diffie-Hellman (q-SDH) assumption [38, 71] which is at most as hard as breaking or solving the discrete logarithm (DL) problem. The DL problem is known to be breakable by quantum adversaries. In particular, Shor's algorithm [196] can be used to break this assumption in polynomial time with quantum computers. Hence, the EPID scheme used by SGX is not quantum-resistant.

We will now survey the EPID schemes proposed in the literature whose security is based on quantum-resistant assumptions. To the best of our knowledge, there are two classes of quantum-resistant assumptions on which the proposed EPID or EPID-like schemes are based: symmetric primitives (which can be obtained from arbitrary one-way functions) and lattice-based assumptions.

6.3.2 Construction based on one-way functions

The EPID scheme in [54] follows a variant of the generic construction for group signatures provided by [33] and instantiates the generic construction based on the BBS+ signature scheme [38, 39, 58, 23] which relies on the q-SDH assumption. The generic construction [33] for group signatures is built on public-key encryption scheme, non-interactive zero-knowledge (NIZK) proofs for NP and a digital signature scheme, with the public-key encryption used in the opening of signatures by the group manager. Boneh et al. [40] adapted the framework of [33] to obtain a non-black box construction of an EPID scheme.[4]

[4] [40] motivate the need for a non-black box construction of EPID from one-way functions due to the result of [4] who show that group signatures imply public-key encryption and hence cannot be based on black-box construction using one-way functions. But as [40] hint in their paper, the construction in [4] crucially relies on the group manager being able to open a group member's signature. It is not clear whether an EPID scheme which does not have this property implies public-key encryption. And hence it is not clear whether a black-box construction is impossible.

EPID scheme of [40] This construction of [40] is the first explicitly stated EPID construction based on symmetric primitives. We will overview their construction here to provide an intuition of an EPID construction. The existing EPID construction(s) [54] revolve around this framework but instantiated with different algebraic assumptions.

Let $\mathsf{S} = (\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify})$ be an existentially unforgeable signature scheme, $\Pi = (P, V)$ be a simulation-sound extractable non-interactive zero-knowledge proof system (NIZK), and a PRF f that also serves as a collision-resistant hash function. (Please refer to [40] or standard cryptography texts for formal definitions of these primitives).

$\mathsf{Setup}(1^\lambda)$ The issuer $\mathcal{I}$ runs the setup as follows:

1. Run $\mathsf{S.Keygen}$ to obtain the verification key and signing key pair $(\mathsf{gpk}, \mathsf{gsk})$.

$\mathsf{Join}\langle(\mathsf{gpk}, \mathsf{gsk}), \mathsf{gpk}\rangle$ The join protocol between the issuer $\mathcal{I}$ and a platform $\mathcal{P}_i$ is as follows:

1. The issuer $\mathcal{I}$ sends a challenge c_i to the platform $\mathcal{P}_i$.
2. $\mathcal{P}_i$ samples $\mathsf{sk}_i \leftarrow \{0,1\}^\lambda$ and sends $t_i = f(sk_i, c_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ sends a signature on (t_i, c_i): $\sigma_i = \mathsf{S.Sign}(\mathsf{gsk}, (t_i, c_i))$.
4. $\mathcal{P}_i$'s secret key is sk_i and let $\mathsf{cert}_i := (t_i, c_i, \sigma_i)$.

$\mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL})$ The signature algorithm is run by a platform $\mathcal{P}_i$ as follows:

1. Sample $r_i \leftarrow \{0,1\}^\lambda$.
2. Let $t = (f(\mathsf{sk}_i, r), r)$.
3. Generate the proof $\pi = \Pi.P(\text{public instance}(\lambda, m, \mathsf{gpk}, t, \mathsf{Sig\text{-}RL}, \mathsf{Key\text{-}RL}),$ private witness$(\mathsf{sk}_i, \mathsf{cert}_i), R)$ for a relation R that we define below. At high level, R checks if (1) t and t' are produced by the same sk_i, (2) σ_i is valid and (3) sk_i does not have signatures in $\mathsf{Sig\text{-}RL}$.
4. Output the signature $\sigma = (t, \pi)$.

The relation R returns true for the witness $(\mathsf{sk}', \mathsf{cert}' = (t', c', \sigma'))$ if

- $t = (f(\mathsf{sk}', r), r)$
- $r \neq c'$
- $\mathsf{S.Verify}(\mathsf{gpk}, (t', c'), \sigma') = 1$
- $t' = f(sk', c')$

- $\forall \sigma_j \in \mathsf{Sig\text{-}RL}, t_{\sigma_j} \neq (f(\mathsf{sk}', r_{\sigma_j}), r_{\sigma_j})$

$\mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Key\text{-}RL}, \mathsf{Sig\text{-}RL}, \sigma)$ A verifier $\mathcal{V}$ outputs 1 if all the following checks hold, else outputs 0.

1. Verify the proof π by checking if $\Pi.V((\lambda, m, \mathsf{gpk}, t, \mathsf{Sig\text{-}RL}, \mathsf{Key\text{-}RL}), \pi) = 1$.
2. $\forall \mathsf{sk}_j \in \mathsf{Key\text{-}RL}$, check $t \neq (f(\mathsf{sk}_j, r), r)$.
3. Check $\sigma \notin \mathsf{Sig\text{-}RL}$.

$\mathsf{Revokesk}(\mathsf{gpk}, \mathsf{Key\text{-}RL}, \mathsf{sk}_i)$ A revocation manager $\mathcal{R}$ updates $\mathsf{Key\text{-}RL}$ as follows:

1. Output $\mathsf{Key\text{-}RL} = \mathsf{Key\text{-}RL} \cup \{\mathsf{sk}_i\}$.

$\mathsf{Revokesig}(\mathsf{gpk}, \mathsf{Sig\text{-}RL}, \sigma, m)$ A revocation manager $\mathcal{R}$ updates $\mathsf{Sig\text{-}RL}$ as follows:

1. Output $\mathsf{Sig\text{-}RL} = \mathsf{Sig\text{-}RL} \cup \{\sigma\}$ if $\mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Key\text{-}RL}, \mathsf{Sig\text{-}RL}, \sigma) = 1$, else output the input $\mathsf{Sig\text{-}RL}$.

Optimization of [40] The above scheme is an EPID scheme satisfying the required security guarantees. But, the resulting signatures sizes are huge in the order of 200 MB for a group size of 2^{30}. [40] identify that the major component of their signatures size is due to the verification circuit of a hash-based signature scheme [160] used by the issuer that is part of the NIZK π. Hence, they propose a second construction where they shorten their signatures using Merkle-consistency proofs [81]. But the group members have to *refresh* their certs often with the $\mathcal{I}$ to remain anonymous. This is motivated by the need for the platforms to interact with the revocation authority $\mathcal{R}$ to update their $\mathsf{Sig\text{-}RL}$. Right now, Intel performs the roles of both $\mathcal{I}$ and $\mathcal{R}$. But, it is not clear how to make this optimization work if these roles are separated.

[40] instantiated their construction with the ZKB++ NIZK protocol [67]. Katz et al. [140] proposed an improved NIZK protocol which leads to a direct improvement in the efficiency of [40]. Both [40] and [140] provide their numbers for the variant using the Merkle proofs optimization. We identify it as an open question to come up with a quantum-resistant signature scheme whose verification circuit admits less AND gates, which usually is an indicator for shorter NIZKs [14]. Table 6.3.3 provides the signature sizes and performance number for these schemes as in their papers. (Both papers do not have their code open-sourced with [140] indicating that they are working towards it). Though these numbers do not exactly correspond to the original EPID definition of [54], these are the only quantum-resistant approaches which can be turned into EPID for

which concrete numbers are available. (As we will see later, the other constructions only provide asymptotic parameter sizes or it is not clear to us how to get an EPID from those constructions). Also, this line of work is based on the "MPC in the head" approach [134, 98] which has shown tremendous progress in the last two years and any new improvement will lead to a more efficient NIZK which in turn leads to shorter signatures and efficient signing.

6.3.3 Construction based on lattices

EPID and EPID-like primitives based on lattice-based assumptions have been studied across the three related primitives that we discussed earlier: group signatures, DAA and BLAC.[5] Here, we provide the state-of-art constructions to the best of our knowledge from each of these lines of work.

Lattice based group signatures Most lattice-based group signatures literature starting from [110] use the traditional definition of group signatures with the group manager being able to trace group members' signatures. Bootle et al. [47] started the study of "fully dynamic" group signatures where a group member can enter and leave (or be revoked from) the group at any instant and more importantly, a separation between the group manager and a tracing authority with the anonymity property extended to the group manager (but not the tracing authority, of course). Ling et al. [153] achieve the best key and signature sizes which grow logarithmically in the size of the group in this model. Very recently, del Pino et al. [79] provide the most efficient lattice-based (or even the most efficient quantum-resistant) group signature scheme for larger group sizes in terms of concrete parameters. It is another interesting open question to convert their scheme to provide provable anonymity against a malicious group manager.

Lattice-based DAA Kassem et al. [138] proposed a lattice-based DAA scheme using Boyen's signature scheme [48] and Baum et al's commitment scheme [31]. Their scheme does not provide a way to obtain signature-based revocation.[6]

[5]Unfortunately, these works do not compare with the works from the other two primitives.

[6]The eprint version of the paper had variables explained across different corners of the paper and sometimes overloaded. We tried our best to provide the accurate results.

Table 6.1: Post-quantum EPID and EPID-like schemes.

Scheme	Type	Group size	Signature size	Signing time
[40]	EPID optimized	2^{10}	1.85 MB	–
[140]	EPID optimized	2^{10}	418 KB	3.0 s
[140]	EPID optimized	2^{13}	532 KB	3.8 s
[153]	F.D. group sig	2^{ℓ}	$\tilde{O}(\lambda\ell)$	–
[79]	group sig	2^{80}	581 KB	0.4 s
[138]	DAA	2^{ℓ}	$cO(n)[k(m'+1)+km(2\ell+2)]$ polynomials in $\mathcal{R}_q$	$2m+m'+1$ polynomial mults

NOTES: F.D. corresponds to fully dynamic. In [138], c is the number of rounds for the proof to be repeated, $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n+1\rangle$, $m = O(\log q)$, $m' < m$ and $k < \lfloor \log q \rfloor + 1$.

Lattice-based BLAC Yang et al. [209] construct "weak PRFs"[7] based on the learning with rounding problem [28] which admit efficient Stern's protocol [198] to prove various zero-knowledge arguments on the lattice problem. They use this primitive to construct various privacy-enhancing applications including a BLAC scheme. Their BLAC protocol instantiates the generic framework that we described earlier using the signature scheme of [151] and their weak PRF scheme instead of the regular PRF. Unfortunately, they do not have concrete parameters or numbers for their instantiation.

6.3.4 Quantum security

Till now we discussed the construction for EPID (and EPID-like primitives) based on quantum-resistant assumptions. All the discussed constructions are secure against adversaries who have access to quantum computers and run the EPID algorithms with "classical" inputs. But this does not guarantee security against adversaries who run these algorithms with "quantum" inputs i.e. a superposition of inputs. While [40] use Unruh's transform [204] instead of the classical Fiat-Shamir transform [87] to prove their security in the quantum random oracle model, it is not clear whether the overall security of the EPID scheme will hold when an adversary has a quantum oracle to all the EPID algorithms. The same question is also unanswered for the lattice based constructions. Boneh and Zhandry [46] studied this for signature schemes where an adversary can issue superposition of messages as queries in the EUF-CMA game. It is an interesting question to discuss for more expressive signature schemes like EPID.

[7]Their weak PRF has the property that for (almost) every input output pair there exists at most one secret key that can evaluate the input to the output.

6.4 AES and MAC

SGX uses a slightly modified AES-CTR mode for encryption and Carter-Wegman MAC algorithm. We will now discuss the security of these primitives under the stronger quantum security model where the adversary can make superposed queries to the cryptographic algorithms.

For the block-cipher based primitives, Grover's algorithm [116] provides a square-root attack. The algorithm provides a quadratic speedup to an adversary who has black-box access to the function of interest which allows superposed queries. Hence, the recommendation is to double the key size. That is, one should use AES-256 to provide a 128 bit quantum security. Anand et al. [18] also proved that CTR mode is secure i.e. IND-qCPA security [46] assuming that the underlying blockcipher is a standard secure PRF. That is we only need AES to be a PRF secure under classical queries for the AES-CTR to be IND-qCPA secure.

For a while, it was widely believed that Grover's algorithm is the best attack possible for any blockcipher based symmetric primitives (encryption, MAC, authenticated encryption). Hence, the only recommendation to make these primitives quantum resistant was believed to be to double the key sizes to guard against [116]. But, further research identified that there is no general truth in this respect. [18] showed that the CBC, CFB, and XTS modes need the underlying blockcipher to be a PRF secure against quantum queries. Worse, Kaplan et al. [137] showed that the most widely used modes of operation for authentication and authenticated encryption like CBC-MAC, PMAC, GMAC, GCM, and OCB can all be broken using only $O(n)$ quantum queries (where n is the size of a block). Fortunately, Boneh and Zhandry [45] provide a slightly modified Wegman-Carter MAC algorithm which is EUF-qCMA secure (EUF-CMA allowing superposed message queries).

6.5 Cryptography for applications in SGX

For an application running inside SGX enclaves to be secure against quantum adversaries, it should also ensure that the cryptographic primitives it uses are quantum-resistant. Standard cryptographic libraries like OpenSSL are ported to SGX and the quantum-resistant algorithms in the library can be used by applications. Open Quantum Safe (OQS) [197] is an ongoing effort to develop *all* quantum-resistant cryptographic algorithms and integrate into widely used protocols and libraries. When these are ported to SGX, applications can use these algorithms for the cryptography they need.

6.6 Conclusion

Some of the core cryptographic primitives used by SGX are not secure against quantum adversaries. The EPID primitive which is a vital cog for remote attestation and hence entire SGX still requires further research for a practical quantum-resistant construction to be obtained. The changes in the implementation of these core primitives might require a few years to be integrated since some of these constructions like the lattice-based ones are relatively new to the hardware community. Intel would need a few years to design a fast and side-channel resistant version of the quantum-resistant construction to be incorporated into SGX. Hence, this research area requires more attention too.

Chapter 7

The path ahead

A design for enabling controlled computations over encrypted data can be evaluated in terms of three parameters: security assumptions required, functionality supported and performance achieved.[1] The three designs proposed in this thesis target different goals in the tradeoff between these three factors as summarized at a high level in Figure 7.1. We will recap the current state of these designs and identify future research directions.

7.1 Cryptographic constructions

The first category consists of the designs whose security is based on standard cryptographic assumptions.

Solution proposed in this thesis

- *Security* - Well-studied lattice-based assumptions.
- *Functionality* - Programs that can be represented as boolean circuits or branching programs.
- *Performance* - Not practical now for reasonable functionalities. A secure program evaluation involves an ABE decryption of the FHE evaluation of the program.

[1] There is also a fourth factor of the ease and the cost of deployment which plays an important role in industry applications. But we will avoid that for this discussion since we believe that there are many hidden factors that influence and justify the *ease* and the *cost*.

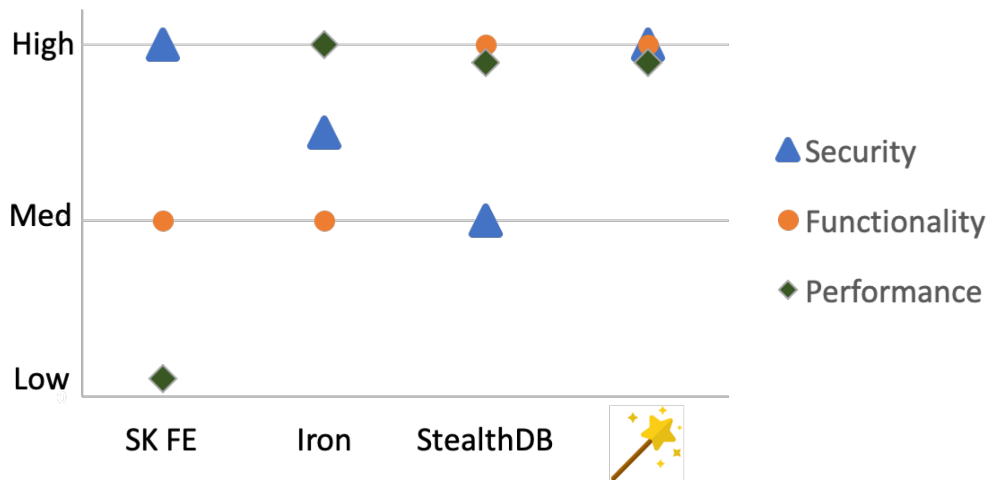


Figure 7.1: Summary of designs in this thesis.

Next steps We require faster core lattice algorithms which admit shorter parameters (for eg. a faster trapdoor sampling which admits small trapdoor sizes). Some recent work including [123] have made significant progress in this direction. We also need to address some fundamental questions about supporting multiple (possibly pre-determined number of) keys. [106] constructs multi-key FE scheme without short secret keys with all their parameters growing with the number of keys supported. The ideas from this scheme might help.

If the property of having short secret keys is not important for the application, the design of [106] will be a more efficient FE design than ours. The performance bottleneck in their construction involves garbling and evaluating universal circuits. The avenue for future research here is to optimize garbled circuits for universal circuits.

7.2 FE from trusted hardware

This category proposes practical designs with an additional assumption of the existence of trusted hardware.

Solution proposed in this thesis

- *Security* - Relies on the security of the trusted hardware used. In the case of SGX, we need to trust Intel, their SGX implementation and their closed source x86 architecture.
- *Functionality* - Programs that are modified to resist all the side-channel attacks that the trusted hardware is subject to.
- *Performance* - Operate at native processor speeds for smaller datasets. Paging is required for larger datasets. Also the overhead incurred in side-channel proofing.

Next steps To protect against memory side-channels without modifications to the applications, we require faster Oblivious RAM [101] constructions. There are newer cache-based attacks on SGX due to its tight coupling with the x86 architecture [56]. It is an interesting research direction to find safeguards to these attacks.

Other trusted hardware designs need exploration and development. AMD introduced Secure Encrypted Virtualization (SEV) [136], but severe attacks have been shown [126, 166, 82]. On the other hand, the "Keystone Enclave" project [144] led by MIT and Berkeley develops an end-to-end open source implementation of a trusted hardware using the Sanctum design [76] based on the RISC-V architecture. Sanctum is among the most-secure trusted hardware designs available against software attacks (see [75] for a comparison between the trusted hardware designs). Google and Microsoft are working on "open enclave" projects for an easier porting and interface with various hardware designs. Designing secure applications on top of these will be an important research direction to explore once these projects mature.

7.3 Encrypted databases using trusted hardware supporting complete SQL

The third category strives for secure designs supporting a complete SQL functionality with practical performance.

Solution proposed in this thesis

- *Security* - A similar reliance on the trusted hardware used. Encrypted databases are also subjected to access pattern leakages against persistent adversaries. No protection

against malicious adversaries unless a bulk of the DBMS is run inside the trusted memory region.

- *Functionality* - Complete SQL. (That's the type of databases we are looking at!)
- *Performance* - Practical performance when operating over datasets around 5 GBs.

Next steps An interesting research direction is to study the applicability of differential privacy like techniques to access patterns. This would help us provably reduce the information leakage through access patterns.

The limited size of the trusted hardware memory is a hindrance to scaling. The main bottleneck here is the maintenance of the integrity tree for the trusted memory region. A fundamental research question here is to come up with a practical integrity (and freshness) protection mechanism that is possibly aware of the hardware architecture.

References

[1] TPC-C benchmark. http://www.tpc.org/tpcc/, 1992.

[2] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In *TCC*, pages 480–497, 2010.

[3] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In *PKC*, pages 733–751, 2015.

[4] Michel Abdalla and Bogdan Warinschi. On the minimal assumptions of group signature schemes. In *ICICS*, pages 1–13, 2004.

[5] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Efficient lattice (H)IBE in the standard model. In *EUROCRYPT*, pages 553–572, 2010.

[6] Shweta Agrawal, David Mandell Freeman, and Vinod Vaikuntanathan. Functional encryption for inner product predicates from learning with errors. In *ASIACRYPT*, pages 21–40, 2011.

[7] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In *CRYPTO*, pages 500–518, 2013.

[8] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In *CRYPTO III*, pages 333–362, 2016.

[9] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *STOC*, pages 99–108, 1996.

[10] Miklós Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In *STOC*, pages 10–19, 1998.

[11] Miklós Ajtai. Generating hard instances of the short basis problem. In *ICALP*, pages 1–9, 1999.

[12] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.

[13] Joseph A. Akinyele, Matthew W. Pagano, Matthew D. Green, Christoph U. Lehmann, Zachary N. J. Peterson, and Aviel D. Rubin. Securing electronic medical records using attribute-based encryption on mobile devices. In *SPSM*, pages 75–86, 2011.

[14] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT I*, pages 430–454, 2015.

[15] Tiago Alves and Don Felton. ARM trustzone. *Information Quarterly*, 3(4):18–24, 2004.

[16] Joël Alwen, Manuel Barbosa, Pooya Farshim, Rosario Gennaro, S. Dov Gordon, Stefano Tessaro, and David A. Wilson. On the relationship between functional encryption, obfuscation, and fully homomorphic encryption. In *IMACC*, pages 65–84, 2013.

[17] Amazon. AWS shell interface specification. https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md, 2017. Accessed: 2017-10-01.

[18] Mayuresh Vivekanand Anand, Ehsan Ebrahimi Targhi, Gelo Noel Tabia, and Dominique Unruh. Post-quantum security of the CBC, CFB, OFB, CTR, and XTS modes of operation. In *PQCrypto*, pages 44–63, 2016.

[19] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *CRYPTO I*, pages 308–326, 2015.

[20] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[21] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof

Fetzer. SCONE: secure linux containers with Intel SGX. In *OSDI*, pages 689–703, 2016.

[22] Nuttapong Attrapadung, Benoît Libert, and Elie de Panafieu. Expressive key-policy attribute-based encryption with constant-size ciphertexts. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC*, volume 6571 of *LNCS*, pages 90–108. Springer, 2011.

[23] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In *SCN*, pages 111–125, 2006.

[24] Daniel Augot, Lejla Batina, Daniel J. Bernstein, Joppe Bos, Johannes Buchmann, Wouter Castryck, Orr Dunkelman, Tim Güneysu, Shay Gueron, Andreas Hülsing, Tanja Lange, Mohamed Saied Emam Mohamed, Christian Rechberger, Peter Schwabe, Nicolas Sendrier, Frederik Vercauteren, and Bo-Yin Yang. Initial recommendations of long-term secure post-quantum systems. https://pqcrypto.eu.org/docs/initial-recommendations.pdf, 2015.

[25] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. In *FC*, 2017.

[26] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216, 2011.

[27] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A security analysis of amazon's elastic compute cloud service. In *SAC*, pages 1427–1434, 2012.

[28] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT*, pages 719–737, 2012.

[29] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *EuroS&P*, pages 245–260, 2016.

[30] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. In Juris Hartmanis, editor, *STOC*, pages 1–5. ACM, 1986.

[31] Carsten Baum, Ivan Damgård, Vadim Lyubashevsky, Sabine Oechsner, and Chris Peikert. More efficient commitments from structured lattice assumptions. In *SCN*, pages 368–385, 2018.

[32] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, pages 267–283, 2014.

[33] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *EUROCRYPT*, pages 614–629, 2003.

[34] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, pages 531–545, 2000.

[35] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *FOCS*, pages 171–190, 2015.

[36] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal on Computing*, 32(3):586–615, 2003.

[37] D. Boneh, A. Sahai, and B. Waters. Functional encryption: a new vision for public-key cryptography. *Commun. ACM*, 55(11):56–64, 2012.

[38] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *EUROCRYPT*, pages 56–73, 2004.

[39] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, pages 41–55, 2004.

[40] Dan Boneh, Saba Eskandarian, and Ben Fisch. Post-quantum group signatures from symmetric primitives. *IACR Cryptology ePrint Archive*, 2018:261, 2018.

[41] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[42] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In *EUROCRYPT*, pages 533–556, 2014.

[43] Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors. *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. ACM, 2013.

[44] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *TCC*, pages 253–273, 2011.

[45] Dan Boneh and Mark Zhandry. Quantum-secure message authentication codes. In *EUROCRYPT*, pages 592–608, 2013.

[46] Dan Boneh and Mark Zhandry. Secure signatures and chosen ciphertext security in a quantum computing world. In *CRYPTO II*, pages 361–379, 2013.

[47] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. Foundations of fully dynamic group signatures. In *ACNS*, pages 117–136, 2016.

[48] Xavier Boyen. Lattice mixing and vanishing trapdoors: A framework for fully secure short signatures and more. In *PKC*, pages 499–517, 2010.

[49] Xavier Boyen. Attribute-based functional encryption on lattices. In *TCC*, pages 122–142. Springer, 2013.

[50] Zvika Brakerski and Gil Segev. Function-private functional encryption in the private-key setting. In *TCC II*, pages 306–324, 2015.

[51] Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based FHE as secure as PKE. In Moni Naor, editor, *ITCS*, pages 1–12. ACM, 2014.

[52] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017.

[53] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS*, pages 132–145, 2004.

[54] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *IJIPSI*, 1(1):3–33, 2011.

[55] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: when elasticity snaps back. In *CCS*, pages 389–400, 2011.

[56] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[57] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *IEEE SP*, pages 901–920, 2017.

[58] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, pages 56–72, 2004.

[59] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[60] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In *TCC II*, pages 468–497, 2015.

[61] D. Cash, D. Hofheinz, E. Kiltz, and C. Peikert. Bonsai trees, or how to delegate a lattice basis. *J. Cryptology*, 25(4):601–639, 2012.

[62] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*, 2014.

[63] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO I*, pages 353–373, 2013.

[64] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *EUROCRYPT*, pages 351–368, 2014.

[65] David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In *HPCA*, pages 1–12, 2010.

[66] Nishanth Chandran, Vipul Goyal, Aayush Jain, and Amit Sahai. Functional encryption: Decentralised and delegatable. Cryptology ePrint Archive, Report 2015/1017, 2015. http://eprint.iacr.org/2015/1017.

[67] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *CCS*, pages 1825–1842, 2017.

[68] David Chaum and Eugène van Heyst. Group signatures. In *EUROCRYPT*, pages 257–265, 1991.

[69] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, pages 645–658, 2017.

[70] Yilei Chen, Craig Gentry, and Shai Halevi. Cryptanalyses of candidate branching program obfuscators. In *EUROCRYPT*, pages 278–307, 2017.

[71] Jung Hee Cheon. Security analysis of the strong Diffie-Hellman problem. In *EUROCRYPT*, pages 1–11, 2006.

[72] Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. Functional encryption from (small) hardware tokens. In *ASIACRYPT II*, pages 120–139, 2013.

[73] Clifford Cocks. An identity based encryption scheme based on quadratic residues. In Bahram Honary, editor, *IMA*, volume 2260 of *LNCS*, pages 360–363. Springer, 2001.

[74] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In Ran Canetti and Juan A. Garay, editors, *CRYPTO I*, volume 8042 of *LNCS*, pages 476–493. Springer, 2013.

[75] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:086, 2016.

[76] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, pages 857–874, 2016.

[77] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.

[78] Verizon data breach incident report. https://regmedia.co.uk/2016/05/12/dbir_2016.pdf, 2016.

[79] Rafaël del Pino, Vadim Lyubashevsky, and Gregor Seiler. Lattice-based group signatures and zero-knowledge proofs of automorphism stability. In *CCS*, pages 574–591, 2018.

[80] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139, 2008.

[81] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure logging schemes and certificate transparency. In *ESORICS II*, pages 140–158, 2016.

[82] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *CoRR*, abs/1712.05090, 2017.

[83] Muhaimin Dzulfakar. Advanced mysql exploitation. Black Hat Las Vegas, 2009.

[84] Keita Emura, Atsuko Miyaji, Akito Nomura, Kazumasa Omote, and Masakazu Soshi. A ciphertext-policy attribute-based encryption scheme with constant ciphertext length. In Feng Bao, Hui Li, and Guilin Wang, editors, *ISPEC*, volume 5451 of *LNCS*, pages 13–23. Springer, 2009.

[85] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[86] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *ESORICS II*, pages 123–145, 2015.

[87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[88] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, pages 3–8. ACM, 2012.

[89] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with SGX. In *DBSec*, pages 386–408, 2017.

[90] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *IEEE SP*, pages 172–191, 2017.

[91] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *HotOS*, 2005.

[92] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 1–17. Springer, 2013.

[93] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, pages 40–49, 2013.

[94] Sanjam Garg, Craig Gentry, Shai Halevi, Amit Sahai, and Brent Waters. Attribute-based encryption for circuits from multilinear maps. In Ran Canetti and Juan A. Garay, editors, *CRYPTO II*, volume 8043 of *LNCS*, pages 479–499. Springer, 2013.

[95] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[96] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC II*, volume 9015 of *LNCS*, pages 498–527. Springer, 2015.

[97] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Cynthia Dwork, editor, *STOC*, pages 197–206. ACM, 2008.

[98] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, pages 1069–1083, 2016.

[99] Henri Gilbert, editor. *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *LNCS*. Springer, 2010.

[100] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, 1987.

[101] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[102] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC*, 2008.

[103] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In *EUROCRYPT 2014*, pages 578–602, 2014.

[104] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Boneh et al. [43], pages 555–564.

[105] Google. Encrypted BigQuery client. `https://github.com/google/encrypted-bigquery-client`, 2017.

[106] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 162–179. Springer, 2012.

[107] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Attribute-based encryption for circuits. In Boneh et al. [43], pages 545–554.

[108] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Predicate encryption for circuits from LWE. In Rosario Gennaro and Matthew Robshaw, editors, *CRYPTO II*, volume 9216 of *LNCS*, pages 503–523. Springer, 2015.

[109] Sergey Gorbunov, Vinod Vaikuntanathan, and Daniel Wichs. Leveled fully homomorphic signatures from standard lattices. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *STOC*, pages 469–477. ACM, 2015.

[110] S. Dov Gordon, Jonathan Katz, and Vinod Vaikuntanathan. A group signature scheme from lattice assumptions. In *ASIACRYPT*, pages 395–412, 2010.

[111] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, pages 308–326, 2010.

[112] Vipul Goyal, Abhishek Jain, Venkata Koppula, and Amit Sahai. Functional encryption for randomized functionalities. In *TCC II*, pages 325–351, 2015.

[113] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *CCS*, pages 89–98. ACM, 2006.

[114] Patrick Grofig, Isabelle Hang, Martin Härterich, Florian Kerschbaum, Mathias Kohler, Andreas Schaad, Axel Schröpfer, and Walter Tighzert. Privacy by encrypted databases. In *Annual Privacy Forum*, pages 56–69. Springer, 2014.

[115] Trusted Computing Group. Trusted platform module. https://trustedcomputinggroup.org/, 2009.

[116] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC*, pages 212–219, 1996.

[117] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331, 2018.

[118] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *ACM CCS*, pages 1353–1364, 2016.

[119] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*, pages 162–168, 2017.

[120] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Security & Privacy*, 14(6):54–62, 2016.

[121] Bernardo Damele Assumpcao Guimaraes. Advanced sql injection to operating system full control. Black Hat Europe, 2009.

[122] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin R. B. Butler, and Patrick Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *FC Workshops*, pages 302–318, 2016.

[123] Kamil Doruk Gür, Yuriy Polyakov, Kurt Rohloff, Gerard W. Ryan, and Erkay Savas. Implementation and evaluation of improved gaussian sampling for lattice trapdoors. *IACR Cryptology ePrint Archive*, 2017:285, 2017.

[124] Shai Halevi and Victor Shoup. Algorithms in HElib. In *CRYPTO I*, pages 554–571, 2014.

[125] Ishay Haviv and Oded Regev. Tensor-based hardness of the shortest vector problem to within almost polynomial factors. In *STOC*, pages 469–477, 2007.

[126] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *VEE*, pages 129–142, 2017.

[127] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, pages 533–549, 2016.

[128] Intel. Intel Trusted Execution Technology, 2009.

[129] Intel. Intel software guard extensions programming reference. 2016.

[130] Intel. SGX documentation: `sgx_create_monotonic_counter`. https://software.intel.com/en-us/node/696638, 2016.

[131] Intel. SGX documentation: `sgx_get_trusted_time`. https://software.intel.com/en-us/node/696636, 2016.

[132] Intel. Intel SGX version 2. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf, 2017. Accessed: 2017-02-16.

[133] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *CT-RSA*, pages 90–107, 2016.

[134] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM J. Comput.*, 39(3):1121–1152, 2009.

[135] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel software guard extensions: Epid provisioning and attestation services. 2016.

[136] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. Whitepaper, 2016.

[137] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking symmetric cryptosystems using quantum period finding. In *CRYPTO II*, pages 207–237, 2016.

[138] Nada E. L. Kassem, Liqun Chen, Rachid El Bansarkhani, Ali El Kaafarani, Jan Camenisch, and Patrick Hough. L-DAA: lattice-based direct anonymous attestation. *IACR Cryptology ePrint Archive*, 2018:401, 2018.

[139] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, pages 115–128, 2007.

[140] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In *CCS*, pages 525–537, 2018.

[141] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*, pages 146–162, 2008.

[142] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[143] Subhash Khot. Hardness of approximating the shortest vector problem in lattices. *J. ACM*, 52(5):789–808, 2005.

[144] Dayeol Lee, Sagar Karandikar, Srini Devadas, Krste Asanovic, Albert Ou, Dawn Song, and Ilia Lebedev. Keystone: Open-source secure hardware enclave. https://keystone-enclave.org/, 2018. Accessed: 2018-10-31.

[145] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[146] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[147] Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In *CCS*, pages 981–992, 2016.

[148] Allison B. Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In Gilbert [99], pages 62–91.

[149] Allison B. Lewko and Brent Waters. New techniques for dual system encryption and fully secure HIBE with short ciphertexts. In Daniele Micciancio, editor, *TCC*, volume 5978 of *LNCS*, pages 455–479. Springer, 2010.

[150] M. Li, S. Yu, Y. Zheng, K. Ren, and W. Lou. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):131–143, 2013.

[151] Benoît Libert, San Ling, Fabrice Mouhartem, Khoa Nguyen, and Huaxiong Wang. Signature schemes with efficient protocols and dynamic group signatures from lattice assumptions. In *ASIACRYPT II*, pages 373–403, 2016.

[152] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177, 2000.

[153] San Ling, Khoa Nguyen, Huaxiong Wang, and Yanhong Xu. Lattice-based group signatures: Achieving full dynamicity with ease. In *ACNS*, pages 293–312, 2017.

[154] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018.

[155] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, pages 87–101, 2015.

[156] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *ACM CCS*, pages 311–324, 2013.

[157] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution. Cryptology ePrint Archive, Report 2017/048, 2017. http://eprint.iacr.org/2017/048.

[158] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[159] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, page 10, 2013.

[160] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[161] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, 2000.

[162] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *EUROCRYPT*, pages 700–718, 2012.

[163] Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In Leonard J. Schulman, editor, *STOC*, pages 351–358. ACM, 2010.

[164] Microsoft SQL Server 2016. Always encrypted database engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx, 2017.

[165] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In *CRYPTO*, 2016.

[166] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In *EuroSec*, pages 1:1–1:6, 2018.

[167] Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16(5):38–41, 2018.

[168] Muhammad Naveed, Shashank Agrawal, Manoj Prabhakaran, XiaoFeng Wang, Erman Ayday, Jean-Pierre Hubaux, and Carl A. Gunter. Controlled functional encryption. In *CCS*, pages 1280–1291, 2014.

[169] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM CCS*, pages 644–655, 2015.

[170] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. Hop: Hardware makes obfuscation practical. In *NDSS*, 2017.

[171] NIST. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, 2017.

[172] NSA. Cryptography today. https://www.nsa.gov/ia/programs/suiteb_cryptography/, 2015.

[173] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.

[174] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *EuroSys*, pages 238–253, 2017.

[175] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *OSDI*, pages 587–602, 2016.

[176] J. Papanis, S. Papapanagiotou, A. Mousas, G. Lioudakis, D. Kaklamani, and I. Venieris. On the use of attribute-based encryption for multimedia content protection over information-centric networks. *Transactions on Emerging Telecommunications Technologies*, 25(4):422–435, 2014.

[177] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steven M. Bellovin. Blind seer: A scalable private DBMS. In *IEEE SP*, pages 359–374, 2014.

[178] Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, pages 422–439, 2012.

[179] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In *EUROCRYPT*, 2017.

[180] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In *STOC*, pages 333–342, 2009.

[181] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[182] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[183] PostgreSQL 9.5.10 Documentation. Extensions. https://www.postgresql.org/docs/9.5/static/external-extensions.html, 2018. Accessed: 2018-01-29.

[184] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *IEEE SP*, pages 264–278, 2018.

[185] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, pages 431–446, 2015.

[186] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.

[187] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[188] Phillip Rogaway. Authenticated-encryption with associated-data. In *ACM CCS*, pages 98–107, 2002.

[189] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 457–473. Springer, 2005.

[190] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *IEEE SP*, pages 38–54, 2015.

[191] Edward J. Schwartz, David Brumley, and Jonathan M. McCune. Contractual anonymity. In *NDSS*, 2010.

[192] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. *CoRR*, abs/1702.08719, 2017.

[193] Jaebaek Seo, Byoungyoung Lee, Sungmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[194] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[195] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB linux applications with SGX enclaves. In *NDSS*, 2017.

[196] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[197] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the Open Quantum Safe project. In *SAC*, pages 14–37, 2016.

[198] Jacques Stern. A new paradigm for public key identification. *IEEE Trans. Information Theory*, 42(6):1757–1768, 1996.

[199] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171, 2003.

[200] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. Aegis: A single-chip secure processor. *IEEE Design & Test of Computers*, 24(6):570–580, 2007.

[201] Chia-che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *EuroSys*, pages 9:1–9:14, 2014.

[202] Chia-che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *EuroSys 2014*, pages 9:1–9:14, 2014.

[203] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blacklistable anonymous credentials: blocking misbehaving users without ttps. In *CCS*, pages 72–81, 2007.

[204] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. In *EUROCRYPT II*, pages 755–784, 2015.

[205] Brent Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 619–636. Springer, 2009.

[206] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[207] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS I*, pages 440–457, 2016.

[208] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, pages 640–656, 2015.

[209] Rupeng Yang, Man Ho Au, Junzuo Lai, Qiuliang Xu, and Zuoxia Yu. Lattice-based techniques for accountable anonymity: Composition of abstract Stern's protocols and weak PRF with efficient protocols from LWR. *IACR Cryptology ePrint Archive*, 2017:781, 2017.

[210] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

[211] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.