FIFE: A Framework for Investigating Functional Encryption

by

Gaurav Singh

S.B., Mathematics and EECS, Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Arkady Yerukhimovich
Research Staff, MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Shafi Goldwasser
RSA Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

FIFE: A Framework for Investigating Functional Encryption

by

Gaurav Singh

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In functional encryption, keys are associated with functions, and ciphertexts with messages. Decrypting a message with a key gives the evaluation of the associated function on that message. We look at bounded-collusion functional encryption, where the number of keys for which security is guaranteed is bounded, as it is possible to realize using standard building blocks. For such schemes we aim to understand their practicality for real-world applications.

There are some theoretical constructions of functional encryption, but few implementations. We rectify this by creating the Framework for Investigating Functional Encryption (FIFE). FIFE includes the first implementations for Sahai and Seyalioglu's one-key scheme (CCS 2010), and Gorbunov, Vaikuntanathan, and Wee's bounded-collusion scheme (CRYPTO 2012), and is easily extendable. We used FIFE to evaluate their performance, and to measure the impact of using different public-key or secret-key encryption schemes, bounds on collusion, and security levels, for interesting classes of functions.

Thesis Supervisor: Arkady Yerukhimovich
Title: Research Staff, MIT Lincoln Laboratory

Thesis Supervisor: Shafi Goldwasser
Title: RSA Professor of Electrical Engineering and Computer Science

Acknowledgments

First, and foremost, I would like to thank Arkady Yerukhimovich, for his extensive help and guidance over the past year. He has been willing to meet often, and has always been available to answer any questions I have. I have enjoyed working with him, and will look back on it fondly.

I would also like to thank Shafi Goldwasser, for her advice and comments on the direction of the work. She started me off thinking cryptographically; her class was my first experience with these types of questions.

Vinod Vaikuntanathan has provided me with suggestions for what I should investigate, and has always been encouraging. He has also been willing to meet on short notice, and always been helpful, and for this I am grateful.

I would also like to thank Ranjit Kumaresan for helpful discussions, and pushing me to clearly articulate the motivations for different parts of our work.

I would also like to thank Tej Kanwar for listening to me discuss the same problems over and over, helping me learn C++, helping me debug C++, and always encouraging me to continue to work, especially when writing this thesis.

Finally, I would also like to thank Sahiba Singh and Corinn Herrick for giving me helpful feedback as I wrote this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

By now, most people are familiar with the concept of public-key encryption. If Alice publishes a public key associated with a secret key which she doesn't share, anyone who has the public key can encrypt data of their choosing, such that only Alice can recover the data, even if others see the encrypted message [59]. This cryptographic primitive, along with related work, has been widely used to create secure online communication, online financial transactions, and private data storage, that enables much of the current tech industry.

These uses of public-key cryptography allow us to control exactly who can access our information. Given the breadth of applications, it may seem like public-key cryptography allows us to control our data in any way we want. However, it has the disadvantage that it is all-or-nothing; given an encrypted message, one either can decrypt it and access the plaintext, or learn nothing about it. This does not allow a recipient to only learn specific features of the data. We can think of these features as a function applied to the data.

One example use case is secure email. Suppose Alice is emailing Bob, who is using a commercial email service that offers spam filtering. Now, suppose Bob wants to only receive encrypted email, so that only he can read it. Using only public-key encryption, if he wanted his email service to continue to provide spam filtering, he would have to allow the email provider to decrypt the messages, so it can check if they are spam. Ideally, Bob would not have to give his email provider access to all of

his email.

Another such use case is for researchers accessing sensitive data, as discussed by Naveed et al. [54]. Researchers may be interested in doing research on genomic and other personal data. However, many people may be uncomfortable revealing this information unless they know that their data is only used for research purposes. Allowing only certain analyses about each person's data ensures that personal information is not exposed, and instead only the relevant information for the research is extracted from the encrypted data.

The general idea behind all of these examples is controlled access. We want to be able to have messages encrypted, in a way that still allows us to share a specific, limited amount of information about the message with a third party. Of course, we can do that currently by just revealing the entire message, but ideally we would like to control what information is revealed, and not reveal the entirety of the message.

Functional encryption is a cryptographic primitive which allows us to do this [13]. Using functional encryption, Bob can create a key for a function, such that if he gives Alice the key and encrypted data, she can use both of them to get the value of the function applied to the data. For our first example above, this would allow Bob to encrypt his email, and, using functional encryption, allow the email service provider to only learn whether each email was spam or not, but nothing else about the contents of his email.

In some ways this is similar to the idea of Fully Homomorphic Encryption (FHE). FHE allows computing *any* function on encrypted data to get the encrypted result. However, functional encryption differs from FHE in that when someone computes a function of some encrypted data, in a functional encryption scheme they get the plain text result, and not the encryption of the result an FHE scheme would give. Indeed, we can use fully homomorphic encryption to achieve a functionality similar to functional encryption. Using the parties from the previous example, Bob can encrypt his message under a fully homomorphic encryption scheme, and send it to Alice. She can then use the fully homomorphic scheme to compute the encrypted value of a function of Bob's message. She can then give this to Bob, and ask him to decrypt it

and send it back to her. This variant differs from functional encryption by adding an extra round of interaction between the two parties any time Alice wants to evaluate a function.

Consider the following example, which illustrates why a scheme where additional interaction is not necessary may be desirable. Suppose that Alice is storing her data in a commercial data store, in an encrypted format. The data storage company may require, for copyright (or other legal) reasons, the ability to determine if a user's data violates copyright (or the law). Alice may not feel inclined to cooperate with the company for this, especially if she was unlawfully storing some data. However, if Alice had initially granted them a key to check compliance, this would allow the company to examine the data for copyright (or legal) violations without getting other information.

As opposed to the above construction involving fully homomorphic encryption, there are schemes for functional encryption which do not require the extra interaction between Alice and Bob to get the answer at the end. We discuss these schemes, and their theoretical trade-offs more in Section 3.3. However, while there have been many such schemes created, few implementations exist to evaluate their real-world efficiency.

One of the main reasons that public-key encryption and digital signatures are used on the internet is the existence of efficient protocols and practical implementations. For functional encryption, on the other hand, while there are polynomial time schemes, they may not be as efficient as desired in practice. Moreover, there are few prototype implementations of the known schemes, so it is even less clear how practical or impractical current schemes really are, or how their performance depends on various parameters of the protocol.

In this thesis, we start to address this deficiency. We create a framework in which to experiment with different implementations, and assess their efficiency. In addition, we implement several functional encryption schemes, and do a detailed study of their performance.

1.1 Our Results

We built the Framework for Investigating Functional Investigation (FIFE). FIFE is a library which allows for evaluations of the performance of functional encryption schemes, and comparisons between them. It also allows for measuring the time and space usage impacts of using different building blocks for functional encryption schemes. FIFE also allows investigations of what use cases are practical for a given scheme.

Beyond creating this framework, we, for the first time, implemented and tested a few different bounded-collusion functional encryption schemes, schemes which are secure against an a priori bounded number of key queries. First, we looked at the Sahai-Seyalioglu scheme, which is secure for only one key query [61]. This is both an interesting scheme on it own, and also a useful building block for other schemes. We also looked at the Gorbunov-Vaikuntanathan-Wee (GVW) scheme for bounded-collusion functional encryption [32]. We also compared the GVW scheme to a stateful scheme, where the party issuing keys for functions is allowed to keep track of the keys issued.

For each of these schemes, we investigated their running times, and also the key and ciphertext sizes, in a variety of contexts. We looked into each scheme to find its performance bottlenecks.

For all of the bounded-collusion functional encryption schemes we investigated, encryption plays a starring role. Given this, we evaluated the effect of different public-key and secret-key encryption schemes on the performance of the functional encryption schemes. We evaluated RSA [59], one of the most well known public-key encryption schemes, and AES [1], one of the most common secret-key encryption schemes.

The performance of each scheme also depends on the class of functions it supports. While the schemes studied support all functions, we chose to start our investigations with smaller and simpler classes of functions. We looked at parity functions, which compute the parity of a subset of the input, inner product modulo a prime p, and

Hamming distance. These functions are interesting, while still relatively simple, and serve as a good initial study into the performance characteristics of functional encryption schemes.

We found that for the functional encryption schemes we investigated, running times were generally not much of an issue. However, we found that ciphertext sizes grew very quickly with the complexity of the functions we wanted to calculate, and that for the Gorbunov-Vaikuntanathan-Wee scheme in particular, ciphertext sizes quickly became impractical. For applications where there is one party issuing all the keys, the stateful scheme offers a more practical solution. While functional encryption is very exciting, more work is needed to make it feasible for use.

Chapter 2

Related Work

In this thesis, we aim to understand the efficiency of functional encryption in practice. A lot of work has already gone into defining, designing, and understanding the limits of functional encryption and constructions of it. We review some of that prior work below.

2.1 Functional Encryption and Related Primitives

Functional Encryption was first defined by Boneh, Sahai and Waters in 2011 [13], and by O'Neill in 2010 [55]. However, the idea of only revealing partial information about a ciphertext did not start there. Functional encryption instead was a powerful generalization of many different ideas.

The first of these was identity-based encryption (IBE), first defined by Shamir in 1984 [66]. An IBE scheme is similar to a public-key encryption scheme, but replaces a person's public key with their identity (their identity can be their name, email address, or any other form of identifier). Boneh and Franklin in 2001 defined chosen-ciphertext security for IBE and created an IBE scheme which had chosen-ciphertext security in the random oracle model [12]. Cocks in 2001 created a concrete scheme for this based on the quadratic reciprocity problem, and showed that the communication required was not too large [17]. Boyen and Waters in 2006 created an anonymous hierarchical IBE scheme, in which the random oracle model was not required for security [14].

Attribute-based encryption (ABE) was another special case of functional encryption. First proposed by Sahai and Waters in 2005, ABE allows a user to decrypt data based on attributes that they (or the encrypted data) have [62]. Goyal, Pandey, Sahai, and Waters in 2006 created key-policy ABE, where a ciphertext has a set of attributes, and a key has an access policy which is a monotonic boolean formula (a boolean formula consisting only of AND and OR gates), and the key only decrypts the ciphertext if the attributes of the ciphertext agree with the access policy of the key [35]. In 2010 Lewko, Okamoto, Sahai, Takashima, and Waters created an ABE scheme for monotone boolean formulas which was fully secure, as compared to previous works' weaker proofs of security [50]. In 2013, Gorbunov, Vaikuntanathan, and Wee constructed an ABE scheme for arbitrary polynomial-size circuits, using the learning with errors assumption [33].

Katz, Sahai, and Waters generalized IBE and ABE to predicate encryption, where keys are associated with functions, and a key can decrypt a ciphertext only if its function evaluates to 1 over the attributes of the ciphertext, and created a scheme for inner product predicates [43]. Lewko et al., in the same paper where they created a fully secure ABE scheme, also created a fully secure predicate encryption scheme for inner products [50]. Gorbunov, Vaikuntanathan, and Wee in 2015 created a predicate encryption scheme for arbitrary circuits [34].

Functional encryption is similar to predicate encryption in that we have secret keys associated with functions. However, it differs in that functional encryption can reveal results of these functions, and not just the ciphertext. Boneh et al. in 2011, in addition to defining functional encryption, also defined a simulation based definition of security (defined in more detail in Section 3.2.5), and showed it was impossible to satisfy for a functional encryption scheme for an unbounded number of functions. Agrawal, Gorbunov, Vaikuntanathan, and Wee in 2013 additionally showed that there was a family of circuits for which the even weakest simulation based definition of security didn't hold when an unbounded number of keys are issued [2]. De Caro, Iovino, Jain, O'Neill, Paneth, and Persiano in 2013 continued to study the relationship between various security definitions for functional encryption, and

created a compiler for transforming a functional encryption scheme secure under an indistinguishability based definition (also defined in Section 3.2.5) into one under a form of a simulation based definition [20].

2.1.1 Bounded-Collusion Functional Encryption

When restricted to only issuing a bounded number of keys, the picture is much rosier. Sahai and Seyalioglu in 2010 showed how to use randomized encodings, including garbled circuits, to make a functional encryption scheme secure for issuing one key [61]. Notably, their scheme only needs to assume the existence of public-key encryption, and not anything stronger.

In 2013, Goldwasser, Kalai, Popa, Vaikuntanathan, and Zeldovich showed how to make another functional encryption scheme secure for one key [30]. Their scheme used fully homomorphic encryption, first defined by Rivest, Adleman, and Dertouzos in 1979 [58], and first realized by Gentry in 2009 using methods from lattice cryptography [27]. This scheme had the advantage that the ciphertext sizes did not grow with the size of functions to be computed on them, unlike the previous scheme.

Gorbunov et al. in 2012 also extended any one key scheme to support a bounded number of keys [32]. They also defined bounded-collusion functional encryption, which is secure as long as a party never has keys for more than a bounded number of functions (see Section 3.2.5 for more details). They also showed that their scheme achieved adaptive simulation security, using techniques from non-committing encryption [16], first given by Canetti, Feige, Goldreich, and Naor in 1996. Also, like the Sahai-Seyalioglu scheme, the scheme by Gorbunov et al., using the Sahai-Seyalioglu scheme as a base, only relies on the existence of public-key encryption and pseudorandom generators, and not any stronger assumptions.

In 2016, Agrawal and Rosen made a bounded-collusion functional encryption scheme that was more efficient [3]. This scheme also moved much of the encryption computation offline, so that the encryption algorithm was more efficient. This scheme uses the ring learning with errors problem, and similarities between a cryptosystem based on it [29, 57] and a fully homomorphic cryptosystem [15].

There have also been constructions for functional encryption based on stronger assumptions such as indistinguishability obfuscation or multilinear maps. In 2015, Waters made a functional encryption scheme for an unbounded number of queries using indistinguishability obfuscation that was secure for an indistinguishability-based security definition [72]. Additionally, Garg, Gentry, Halevi, and Zhandry in 2016 showed how to make a functional encryption scheme based on certain assumptions about multilinear maps [26].

2.2 Functional Encryption Implementations

Our work seeks to compare different implementations of functional encryption. While there are not existing implementations of functional encryption, there are closely related implementations. These either use extra interactions, support only a subset of functional encryption, or are limited to only supporting one key.

2.2.1 Ciphertext-Policy Attribute-Based Encryption

In 2007, Bethencourt, Sahai, and Waters implemented and evaluated a ciphertext-policy attribute-based encryption scheme, where a user could only decrypt a ciphertext if their credentials allowed it [10]. As we discussed earlier, this can be thought of as a specific type of functional encryption scheme. However, since their goal was not full functional encryption, they are not limited in the number of users or ciphertexts they can support.

2.2.2 Controlled Functional Encryption

Naveed, Agrawal, Prabhakaran, Wang, Ayday, Hubaux, and Gunter presented Controlled Functional Encryption in 2014 [54]. They defined a new idea of controlled functional encryption, which allows for computations that look similar to functional encryption. Here, there are the same three parties as in functional encryption, a data-owner, an evaluator, and a key authority. However, when an evaluator wants to

evaluate a function, they must ask the key authority for a key for each ciphertext. In comparison, in functional encryption, a key for a function allows you to evaluate that function for every ciphertext, without the additional interaction for each pair of ciphertext and function. Naveed et al. also implemented their proposal, the first for a scheme related to functional encryption.

2.2.3 Function-Hiding Inner Product Functional Encryption

Function-hiding functional encryption means that a key for a function does not reveal the function for which it is a key, and was first proposed by Shen, Shi, and Waters in 2008 [67]. In 2016, Kim, Lewi, Mandal, Montgomery, Roy, and Wu implemented function-hiding functional encryption for inner products [44]. They implemented and optimized constructions by Bishop, Jain, and Kowalczyk [11], and Datta, Dutta, and Mukhopadhyay [19]. Using this, they made a general one-key functional encryption scheme for two inputs, where functions are evaluated over two inputs, as long as the messages encrypted are small. They also implemented their scheme, and evaluated it for vectors of different sizes.

2.3 Secure Computation

Many of the functional encryption schemes we looked at build upon existing tools from secure computation. We discuss existing work and implementations of them for the two ideas that are used in schemes we implement: garbled circuits and the Ben-Or-Goldwasser-Wigderson (BGW) scheme for multiparty computation.

2.3.1 Garbled Circuits

The Sahai-Seyalioglu scheme for functional encryption requires the use of randomized encodings for a circuit [61]. Garbled Circuits are such a randomized encoding, and were discovered by Yao in 1986 [73]. Since then, there have been a number of improvements to the efficiency of garbled circuits, that reduce their computation time

and space usage. Naor, Pinkas, and Sumner in 1999 invented Garbled Row Reduction, which showed how to reduce the space needed per gate by a quarter [53]. In 2008, Kolesnikov and Schneider showed that XOR gates can be computed without any cryptographic operations, called free-XOR, reducing the space and time usages [48]. Pinkas, Schneider, Smart, and Williams in 2009 showed that the Garbled Row Reduction and free-XOR techniques were compatible, and additionally found a technique to reduce the space used by each gate in half, but which was not compatible with free-XOR [56]. In 2015, Zahur, Rosulek, and Evans showed how to combine the two, using half-gates, where each gate is encrypted and decrypted in two steps, to allow both XOR gates that require no cryptographic operations and only half the storage for the rest of the gates [74].

Implementations One of the first implementations of garbled circuits was Fairplay, in 2004 [51]. Pinkas, Schneider, Smart, and Williams in 2009 also made implementations that took advantage of the techniques they found, and compared them [56]. In 2010, TASTY included another garbled circuit which moved some of the setup offline, and analyzed the effects of components on the garbled circuit construction [38]. Huang, Evans, Katz, and Malka in 2011 implemented garbled circuits that could be computed in a streaming manner, removing limits based on what could be stored in memory [39]. Bellare, Hoang, Keelveedhi, and Rogaway in 2013 created JustGarble, further improving the efficiency of evaluating garbled circuits [7]. This was further improved to create libgarble, which incorporated the half-gates technique [36].

2.3.2 The BGW Secure Multiparty Computation Scheme

In 1988, Ben-Or, Goldwasser, and Wigderson created a scheme for secure multiparty computation [9]. This allows many parties to each hold a secret, and collectively compute a function of their secrets, without their secrets being revealed. To do this, they use Shamir's secret sharing scheme [65] to compute an arithmetic circuit over the inputs. Shamir's scheme represents a secret with a polynomial, and uses polynomial interpolation among points of that polynomial to recover the secret. This scheme

allows adding secrets for free, but increases the degree of the polynomial when multiplying. The BGW scheme improves upon this by using additional communication to reduce the degree of the polynomial.

2.4 Necessary Circuit Classes and Universal Circuits

Most functional encryption schemes treat functions as circuits over their inputs. For some of the schemes, we need to evaluate arithmetic operations. These schemes also use universal circuits, which take in a circuit and the input, and evaluate the circuit on the input. In our implementations of functional encryption schemes, we find that reducing the size of these circuits is useful to increase efficiency. We present some of the existing work that finds smaller circuits for these purposes.

2.4.1 Boolean Circuits for Arithmetic Operations

There are many ways to construct boolean circuits for arithmetic operations. In 2009, Kolesnikov, Sadeghi, and Schneider showed how to make such circuits smaller, in a manner well suited to garbled circuits [47]. There are also more asymptotically efficient algorithms for multiplication. Karatsuba's algorithm does so by recursively reducing multiplications to smaller multiplications and additions [41]. The Schönhage-Strassen algorithm is even more efficient, but only practical for larger numbers [63].

2.4.2 Universal Circuits

In 1976, Valiant showed how to make a universal circuit that supports all circuits, with asymptotically optimal size [71]. Kolesnikov and Schneider in 2008 provided an implementation that had smaller constant factors, but was less asymptotically efficient [49]. Kiss and Schneider in 2016 made a universal circuit implementation that achieves Valiant's asymptotic growth, and improved on the constant factors to make it more practical [45].

Chapter 3

Background

In this chapter, we state the notation (in Section 3.1), and define and discuss the building blocks (in Section 3.2), used throughout this thesis. We then move on to discussing different schemes for functional encryption in Section 3.3.

3.1 Notation

We will use $\text{negl}(\kappa)$ to indicate some function f that is negligible in κ, such that for any polynomial p, there is some c such that for $\kappa > c$, $\frac{1}{p(\kappa)} > f(\kappa)$.

We will use $x \xleftarrow{\$} S$ to mean that x is selected uniformly at random from a set S.

We will use $x \leftarrow A$ to mean that x is the output of algorithm A.

We will use $[n] = \{0, 1, \ldots, n-1\}$.

Let $\mathcal{X}_n \subseteq \{0,1\}^n$, and $\mathcal{Y}_m \subseteq \{0,1\}^m$. Then, we call $\mathcal{C} \subseteq \{C : \mathcal{X}_n \to \mathcal{Y}_n\}$ a class of circuits, where for each $C \in C$, given $x \in \mathcal{X}$, it outputs $C(x) \in \mathcal{Y}$. If, for all $C \in \mathcal{C}$, we can describe C using λ bits, we say $|C| = \lambda$, and denote $\mathcal{C} = \mathcal{C}_\lambda$.

3.2 Building Blocks

We define and discuss the theoretical building blocks used in constructions of functional encryption.

3.2.1 Shamir Secret Sharing

Definition 1. An *N-party secret sharing scheme* consists of two algorithms: Share and Reconstruct.

- **Share(x):** Given an input x, output shares of x, $\{s_i\}_{i\in[N]}$.
- **Reconstruct(S):** Given a set S of shares, output y.

Correctness An N-party secret sharing scheme is called *t-reconstructable* if, for any $S \subset \text{Share}(x)$, with $|S| \geq t$, $Reconstruct(S) = x$.

Security An N-party secret sharing scheme is *t-secure* if, for any algorithm A,

$$Pr_{x\in X, S\subset \text{Share}(x), |S|\leq t}[A(S) = x] = \frac{1}{|X|}.$$

In 1979, Shamir created the following N-party, $(t+1)$-reconstructable, t-secure secret sharing scheme based on polynomial interpolation [65].

- **Share(x):** Given an input x in a finite field $\mathbb{F}$, with $|\mathbb{F}| > N$, pick a random degree-t polynomial p over $\mathbb{F}$ such that $p(0) = x$. Now, the shares of x are $j, p(j)$ for $j \in J$, where $J \subset \mathbb{F} \setminus \{0\}$, and $|J| = N$. We denote $j, p(j)$ as $s_j(x)$.
- **Reconstruct(S):** If $S = \{(i, p(i))\}$, and $|S| \geq t+1$, we can do polynomial interpolation to recover p. Then, $p(0) = x$.

Secret sharing is closely related to the idea of secure multiparty computation, where multiple parties each have private inputs, and want to collectively compute a function of their inputs without revealing the inputs. The correctness and security definitions are analogous, just replacing the recovered x with a computed result, $f(x)$. The Ben-Or-Goldwasser-Wigderson (BGW) scheme [9] uses Shamir secret sharing to create a $(t+1)$-reconstructable, t-secure multiparty computation scheme. It does this by noticing the following: in the Shamir secret sharing scheme, if $s_i(x)$ and $s_i(y)$ are

shares of x and y, let,

$$s_i(x+y) = s_i(x) + s_i(y) = p_x(i) + p_y(i) = (p_x + p_y)(i).$$

Now, if $p_{x+y} = p_x + p_y$, we see that $s_i(x+y)$ are shares of p_{x+y}, and we can use the reconstruct algorithm to get $x+y$, with t shares. For multiplication, we note that $s_i(x)s_i(y) = (p_x p_y)(i)$, and this can be reconstructed with $2t+1$ shares.

The BGW scheme starts with the parties all secret sharing their private inputs with everyone else, using Shamir secret sharing. Then, they each use the above observations to compute a function f of their data. The BGW scheme also adds rounds of interaction after multiplications to reduce the degrees of the polynomials, making it t-reconstructable. Without these extra interactions, for a degree D polynomial, the BGW scheme gives an N-party, $(tD+1)$-reconstructable, t-secure multiparty scheme. We will only use this form of the BGW scheme in this thesis. For further details on the interactive component, and detailed proofs of security, see the original paper [9] and a careful treatment by Asharov and Lindell [6].

3.2.2 Traditional Encryption Schemes

We will use *Traditional Encryption Scheme* to mean either a public-key encryption scheme, or a secret-key encryption scheme [59].

Definition 2. A traditional encryption scheme consists of three algorithms.

- **Setup(1^κ):** Given the security parameter κ, generate and output a public key, PK, and a secret key SK.
- **Encrypt(PK, msg):** Given a message msg, and the public key, generate a ciphertext CT for msg.
- **Decrypt(SK, CT):** Given a ciphertext CT for msg, and the secret key SK, output msg.

We abuse the above notation for secret-key encryption by setting $PK = SK$.

Correctness For any $(SK, PK) \leftarrow Setup(1^\kappa)$, we call a traditional encryption scheme correct if,

$$\text{Decrypt}(SK, Encrypt(PK, msg)) = msg$$

always.

Informally, we call a traditional encryption scheme secure if an adversary gains only negligible advantage over randomly guessing in recovering the value of a ciphertext (where for a public-key scheme, the adversary also has access to the public key). There are many variants of security for traditional encryption schemes, and as these details are not important for our work, we leave the details to the presentation by Goldwasser and Micali [31]. The book by Katz and Lindell is another useful reference [42].

3.2.3 Garbled Circuits

Yao originally defined garbled circuits in 1986 as a method of two party secure computation [73]. A garbling scheme is a way to turn a circuit into a method of computing over encrypted data: given a set of inputs, which are only partially known (and partially correspond to another party's secret), it allows you to only learn the known inputs and the output. It does this by encoding, for each input bit, a separate label for 0 or 1.

Definition 3. A *garbling scheme GS* consists of two methods.

- **Garble(C):** Given a circuit $C : \{0,1\}^a \to \{0,1\}^c$, output a garbled circuit G, and labels $l_{i,b}$ for $i \in [a], b \in \{0,1\}$.
- **Evaluate(G, $\{l_{i,x_i}\}_{i\in[a]}$):** Given a garbled circuit G for C, and a labels for $x = (x_i)_{i\in[a]}$, output $C(x)$.

We call a garbling scheme correct if, when $(G, \{l_{i,b}\}_{i\in[\lambda],b\in\{0,1\}}) \leftarrow Garble(C)$,

$$Evaluate(G, \{l_{i,x_i}\}_{i\in[\lambda]}) = C(x).$$

As the security definition will not be directly used in this thesis, we leave the details to Yao's presentation [73]. Gorbunov et al. have a good definition for decomposable randomized encodings, defined by Ishai and Kushilevitz in 2000 [40] and Applebaum, Ishai, and Kushilevitz in 2006 [4], which is a generalization of a garbling scheme [32].

3.2.4 Universal Circuits

Definition 4. Let $\mathcal{C}_\lambda : \mathcal{X}_a \to \mathcal{Y}_c$ be a class of circuits as defined above, where each circuit is described by λ bits. Then, a *universal circuit* U_λ for $\mathcal{C}_\lambda$ is a circuit U_λ such that, on input $x \in \mathcal{X}_a$ and $C \in \mathcal{C}_\lambda$, $U_\lambda(x, C) = C(x) \in \mathcal{Y}_c$.

If $\mathcal{C}_\lambda$ is the set of all possible circuits from $\mathcal{X}_a$ to $\mathcal{Y}_c$, we call U_λ a *general universal circuit.*

In 1976, Valiant showed how to make a general universal circuit, in a way that was asymptotically optimal: that grows with $k \log k$, where k is the size of the input circuit [71]. However, for over 20 years there was not an implementation of general universal circuits, due to concerns about their efficiency in practice. In 2008, Kolesnikov and Schneider made the first implementation, with size about $1.5k \log^2 k$, which is asymptotically slower than Valiant's construction [49]. In 2016, Kiss and Schneider constructed a scheme with the optimal asymptotic efficiency, and showed the performance of their implementation of it on interesting circuits [45].

3.2.5 Functional Encryption

Functional Encryption (FE) was first formally defined by Boneh, Sahai and Waters [13] and O'Neill [55]. The following definition closely resembles that of Gorbunov, Vaikuntanathan, and Wee [32], which in turn is based on that of Boneh et al.

Definition 5. Let $\mathcal{C} = \{C_\kappa\}_{\kappa \in \mathbb{N}}$ be a collection of families of circuits. Then, a *functional encryption scheme* for $\mathcal{C}$ consists of four algorithms, Setup, KeyGen, Encrypt, and Decrypt.

- **Setup(1^κ):** Given the security parameter κ, generate and output a master public key, MPK, and a master secret key MSK.

- **KeyGen(MSK, C):** Given a circuit C, and the master secret key, generate a functional key k_C for the function computed by C.
- **Encrypt(MPK, x):** Given a value x, and the master public key, generate a ciphertext CT for x.
- **Decrypt(FK, CT):** Given a ciphertext CT for x, and the functional key $FK = k_C$ for C, output $C(x)$.

As before, when $MPK = MSK$, we abuse notation to call it a *Secret-key Functional Encryption Scheme (sk-FE).*

We call the functional encryption scheme correct if for all circuits $C \in \mathcal{C}_\kappa$, and all x, if $(MPK, MSK) \leftarrow Setup(1^\kappa)$,

$$Pr\left[Decrypt(KeyGen(MSK, C), Encrypt(MPK, x)) \neq C(x)\right] = \text{negl}(\kappa)$$

where the probability is over the randomness of the Encrypt algorithm.

We will discuss security of Functional Encryption in Section 3.2.5.

Functional encryption, as described above, has three parties, the key authority, the encryptor, and the evaluator, that interact in the following way. The key authority runs the Setup algorithm, and generates the master secret key and master public key. They then announce the master public key to the world. The encryptor uses the master public key to encrypt their message x (or messages) with the Encrypt algorithm and sends the ciphertext(s) CT_x to the evaluator. The evaluator asks the key authority for permission to evaluate a function f (or functions), and the key authority grants permission by running the KeyGen algorithm and issuing the functional key(s) k_f for the function(s). The encryptor and evaluator steps can happen in any order. Then, the evaluator uses the functional key k_f and an encrypted message CT_x to recover the function applied to the message, $f(x)$. Note that here multiple ciphertexts and functional keys can be issued, and cannot depend on one another.

This set of interactions can also happen where the encryptor and key authority are the same party, the setting of sk-FE. Suppose Alice has some data, and she wants

Bob to be able to compute some functions of the data. Then, she can play the roles of the key authority and encryptor, running the Setup, Encrypt, and KeyGen algorithms. This allows her to give Bob ciphertexts for messages, and functional keys for functions. Then, Bob can use these to to evaluate the functions he desires on the ciphertexts he has.

Now, she can encrypt x using Encrypt, and give Bob the resulting ciphertext CT. She can then do this for other values, and also give him other ciphertexts CT'. If Alice publishes the master public key, then anyone else who has it can also generate encrypted inputs. Again, similar to above, note that the ciphertexts must work for all functions, and cannot depend on them.

Just as in traditional encryption schemes, where secret-key encryption can be thought of as a special case of public-key encryption, we can say the same about sk-FE. Now, the only reason secret-key encryption or sk-FE is worth discussing is if it offers some advantages. The first advantage is speed; as we will discuss specifically for RSA and AES in Section 5.1.1, secret-key encryption schemes are much faster in practice. In addition, secret-key encryption primitives are much simpler than public-key ones. sk-FE is interesting for the same reason: it turns out to be faster in practice.

In addition to the presented example, there are many cases where sk-FE is exactly the use case we are looking for, and so it is not less usable than functional encryption. In a functional encryption scheme, the encryptor has to trust the key authority to only give functional keys to functions that reveal limited information, and not the identity function. In sk-FE, they can control this by deciding which functions to issue keys for.

However, there are still situations in which functional encryption is more desirable that sk-FE. Having multiple parties encrypt messages under a single master public key, such that a single functional key allows an evaluator to evaluate a function on all the inputs, can be seen as a good thing. It allows the evaluator to target certain messages while preserving the privacy of the remaining encryptors. More concretely, let the evaluator be an investigative agency, such as the FBI, and the key authority be a cloud storage operator. Now, users can store their data encrypted in the cloud,

but the FBI can still request keys to perform a search on the encrypted data, with a warrant detailing the function required before the cloud storage provider issues the functional key.

Keeping State

In the stateful scheme discussed in Section 3.3.2, we defined a bounded-collusion functional encryption scheme where the key authority must keep state. This relaxation is useful because it allows us to create much more efficient schemes for bounded-collusion functional encryption, as we will discuss in Section 5.3.1. There are many practical applications for which this is a reasonable restriction to place. If there is a central authority issuing all keys, they can keep track of how many functional keys have been issued for a given functional encryption scheme. Also, since they are already storing the master secret key for that scheme, it is not unreasonable for them to also maintain a count of the number of keys issued using that key.

Security of Functional Encryption

Security for functional encryption is not easy to define. There are two main types of security defined: those based on indistinguishability and those based on simulation. Brent, Sahai, and Waters showed that an indistinguishability-based definition of security for functional encryption has some weaknesses, but simulation-based definitions were impossible to achieve with an unbounded number of keys for functions [13].

Indistinguishability-Based Security Defined by O'Neill [55] and Boneh et al. [13], indistinguishability-based security asks an adversary to distinguish between two messages given the ability to ask for secret keys.

Definition 6. We say a functional encryption scheme FE is has *(NA, AD)-IND-security* if for any probabalistic polynomial time (non-adaptive, adaptive) algorithm A, $Pr[b = b'] = \frac{1}{2} + \text{negl}(\kappa)$ for the following experiment.

1. $MPK, MSK \leftarrow FE.Setup(1^\kappa)$

2. $m_0, m_1 \leftarrow A^{FE.KeyGen(MSK,\cdot)}(MPK)$

3. $c \xleftarrow{\$} FE.Encrypt(MSK, m_b)$

4. $b' \leftarrow A^{\mathcal{O}(MSK,\cdot)}(MPK, c)$

Here, we must have that for each k_{C_i} requested by A, in steps 2 or 4, $C_i(m_0) = C_i(m_1)$.

If the adversary is non-adaptive, the oracle $\mathcal{O}$ is the empty oracle, which returns nothing. If the adversary is adaptive, the oracle $\mathcal{O}$ is instead $FE.KeyGen(MSK, \cdot)$.

The condition on the functional keys the adversary is allowed to request is necessary for a meaningful security definition, as a function which differs on the messages will always allow the adversary to identify the challenge ciphertext c.

Boneh et al. showed that this definition is still unsatisfying [13]. They showed that with a circuit family where there are not any circuits C such that $C(x) = C(y)$, having $Encrypt(MPK, x) = x$ can satisfy this definition, even though it leaks more information than is desired.

Simulation Security Boneh et al. instead made a security definition based on simulating functional encryption [13]. Gorbunov et al. slightly broadened their definition to support multiple ciphertexts [32].

Definition 7. A functional encryption scheme FE is adaptively simulation (AD-SIM) secure if, for every probabalistic polynomial time adversary A and simulator S, the outputs of experiment 1,

1. $(MPK, MSK) \leftarrow FE.Setup(1^\kappa)$

2. $x_1, \ldots, x_l \leftarrow A^{FE.KeyGen(MSK,\cdot)}(MPK)$

3. $CT_i \leftarrow FE.Encrypt(MPK, x_i)$ for $1 \leq i \leq l$

4. $\alpha \leftarrow A^{FE.KeyGen(MSK,\cdot)}(MPK, C_1, \ldots, c_l)$

5. Output $(\alpha, x_1, \ldots, x_l)$,

and experiment 2,

1. $(MPK, MSK) \leftarrow FE.Setup(1^\kappa)$
2. $x_1, \ldots, x_l \leftarrow A^{FE.Keygen(MSK,\cdot)}(MPK)$
   - Let $(C_1, \ldots, C_q)$ be A's oracle queries.
   - Let SK_i be the oracle's reply to C_i.
   - Let $\mathcal{V} = \{C_i(x_j) | 1 \leq i \leq q, 1 \leq j \leq l\}$
3. $CT_1, \ldots, CT_l \leftarrow S(MPK, \mathcal{V}, 1^{|x_i|})$
4. $\alpha \leftarrow A^{S^{U_x(\cdot)}(MSK,\cdot)}(MPK, C_1, \ldots, c_l)$
5. Output $(\alpha, x_1, \ldots, x_l)$,

are computationally indistinguishable. Here, $U_x(\cdot)$ is the universal circuit which on input C outputs $C(x)$. Also, S is only allowed to query $U_x(\cdot)$ on its input, C.

If we replace the oracle access in step 4 of each experiment with a null oracle, this is then a non-adaptive functional encryption scheme.

This definition captures the idea that ciphertexts can be simulated only knowing the length of the message. The simulator also has access to the evaluation of the adversary's queries to the oracle on the test inputs. This is necessary, as if the ciphertexts do not agree on the inputs, it is easy for the adversary to distinguish the two experiments. Also, the second stage of the simulator has a universal circuit on the input circuits to again allow the simulator to know what it is trying to simulate.

Unfortunately, while this is clearly a stronger security definition, Boneh. et al. showed that this definition is impossible to satisfy for a functional encryption scheme [13].

Bounded-Collusion Functional Encryption

The result of Boneh et al. [13] rules out the possibility of adaptive simulation security for full functional encryption. However, it does not say anything about the case

where an adversary can only have a limited number of functional keys. This setting is what Gorbunov et al. called bounded-collusion functional encryption [32]. For bounded-collusion functional encryption, the picture is much rosier. They created a bounded-collusion functional encryption scheme (the GVW scheme which we will see below) that had adaptive simulation security.

3.3 Constructions of Functional Encryption

As we discussed in Section 3.2.5, it is impossible to have a functional encryption scheme that has adaptive simulation secure for an unbounded number of function keys. However, it is possible to make schemes that are secure for a bounded number of function keys.

3.3.1 One-key Functional Encryption

The most basic bounded-collusion FE scheme is one which is secure for up to one functional key, or a one-key FE scheme. Such a scheme can issue many functional keys, but is not secure when a person has access to multiple functional keys. In addition, these can serve as a building block for many bounded-collusion schemes. There are multiple existing one-key schemes, each with their own trade-offs. We present some of them here, and discuss their advantages.

Singleton FE

Singleton Functional Encryption, as defined by Gorbunov et al. [32], gives us a functional encryption scheme for the circuit family $\mathcal{C} = \{C\}$, where $C(x) = x$ (since $\mathcal{C}$ has size 1, it does not make sense to talk about one-key or bounded-collusion security levels, as there is only one circuit). This does so in the following way, using a traditional encryption scheme, ES.

- **Setup(1^κ):** Let $(sk_1, pk_1) = ES.Setup(1^\kappa)$, and $(sk_2, pk_2) = ES.Setup(1^\kappa)$ be two key pairs generated by the traditional encryption scheme. Then, output

$(MSK, MPK) = ((sk_1, sk_2), (pk_1, pk_2))$.

- **KeyGen(MSK):** Let $b \xleftarrow{\$} (0, 1)$. Then, for $MSK = (sk_0, sk_1)$, output $FK = (b, sk_b)$.

- **Encrypt(MPK, x):** Output $(ES.Encrypt(pk_0, x), ES.Encrypt(pk_1, x))$ as the ciphertext, for $MPK = (pk_0, pk_1)$.

- **Decrypt(FK, CT):** Output $ES.Decrypt(sk, ct_b)$, for $FK = (b, sk)$ and $CT = (ct_0, ct_1)$.

This can also be thought of as turning a traditional encryption scheme into an interactive non-committing encryption scheme [16]. This will be useful to provide functional encryption schemes with adaptive simulation security.

Sahai-Seyalioglu (SS) Scheme

Sahai and Seyalioglu in 2010 created a one-key functional encryption scheme using public-key encryption, universal circuits, and randomized encodings, which supports any class of circuits [61]. Garbled circuits are a randomized encoding, and so we give the scheme in terms of garbled circuits.

This scheme works as follows, given a base encryption scheme BES, for a circuit family $\mathcal{C}_\lambda$, where λ is the length of the circuit.

- **Setup(1^κ):** Run the base encryption scheme 2λ times. We then output λ pairs of public keys as the MPK, and similarly for secret keys as the MSK.

$$(MSK_{i,j}, MPK_{i,j}) \leftarrow BES.Setup(1^\kappa) \qquad i \in [\lambda], j \in [2]$$

  Output $(MSK, MPK) = (((MSK_{i,0}, MSK_{i,1}))_{i\in[\lambda]}, ((MPK_{i,0}, MPK_{i,1}))_{i\in[\lambda]})$.

- **KeyGen(MSK, C):** For each bit of C, output the corresponding MSK for the base encryption scheme. Output $FK = (MSK_{i,C_i})_{i\in[\lambda]}$.

- **Encrypt(MPK, x):** Let U_x be a universal circuit for $\mathcal{C}_\lambda$. Then, if GS is a garbling scheme, let $(G, \{a_{i,b}\}_{i\in[\lambda],b\in\{0,1\}}) \leftarrow GS.Garble(U_x)$. Now, encrypt each label with the corresponding key, where $MPK = ((MPK_{i,0}, MPK_{i,1}))_{i\in[\lambda]}$.

$$EL_{i,b} = BES.Encrypt(MPK_{i,b}, a_{i,b}) \qquad i \in [\lambda], b \in \{0,1\}$$

  Now, output G and $\{EL_{i,b}\}_{i\in\lambda,b\in\{0,1\}}$.

- **Decrypt(FK, CT):** If $FK = \{MSK_{i,b}\}_{i\in[\lambda]}$, and $CT = G, \{EL_{i,b}\}_{i\in[\lambda],b\in\{0,1\}}$, then let $l_i = BES.Decrypt(MSK_{i,b}, EL_{i,b})$ for $i \in [\lambda]$. Now, for the garbling scheme GS, output $GS.Evaluate(G, \{l_i\}_{i\in[\lambda]})$.

The Base Encryption Scheme can either be a traditional encryption scheme, or the singleton FE scheme, which provides adaptive simulation security. If the singleton FE scheme is used, the above scheme changes the KeyGen step to additionally call KeyGen for the singleton FE scheme.

As we will see in Section 5.1, the Sahai-Seyalioglu scheme leads to large ciphertexts which grow with the size of the circuit, and more specifically, the size of a universal circuit for $\mathcal{C}_\lambda$. However, it has the advantage that it relies only on a traditional encryption scheme, which is well studied and understood.

The Goldwasser-Kalai-Popa-Vaikuntanathan-Zeldovich (GKPVZ) Scheme

In 2013, Goldwasser, Kalai, Popa, Vaikuntanathan, and Zeldovich showed how to create a one-key functional encryption scheme using any fully homomorphic encryption scheme and any attribute-based encryption (ABE) scheme as black boxes [30]. Additionally, their scheme has the property that the ciphertexts do not grow with the size of the circuit, unlike the SS scheme.

This makes use of an ABE scheme where, given $ct = ABE.Encrypt(y, m_0, m_1)$ and $k = ABE.KeyGen(g)$, $ABE.Decrypt(ct, k)$ outputs m_0 if $g(y) = 0$, and else m_1.

- **Setup(1^κ):** This algorithm runs the Setup algorithms on each of the FHE scheme and the ABE scheme.

- **KeyGen(MSK, C):** This algorithm outputs C and n ABE keys, where n is the length of $E(f(x))$ in the FHE scheme. The ABE keys are $ABE.KeyGen(f'_i)$, where for the i-th bit of $E(f(x))$, $f'_i(E(x)) = (f'(E(F(x)))_i = 0)$.

- **Encrypt(MPK, x):** The ciphertext for a message x is x encrypted under an FHE scheme, along with a garbled circuit for the FHE.Decrypt function with the secret key for the FHE scheme included. It also includes the ABE ciphertexts $ABE.Encrypt(x, l_{i,0}, l_{i,1})$, where $l_{i,b}$ is the label for the garbled circuit for the i-th bit, with value b.

- **Decrypt(FK, CT):** This algorithm uses the ABE.Decrypt function with the ABE keys and ciphertexts to get the labels for the garbled circuit. The algorithm now uses these labels to evaluate the garbled circuit to obtain the decrypted value, $f(x)$.

The advantage of this scheme is that the ciphertexts no longer grow with the size of the circuits to be computed. However, this now relies on lattice-based cryptography, fully homomorphic encryption, and attribute-based encryption, all of which are significantly more powerful, and less well studied, constructs than the public-key encryption of the earlier scheme. It would be interesting to explore the trade-off between less efficient components, and the savings of an improved protocol.

3.3.2 Bounded-Collusion Functional Encryption

As with one-key functional encryption schemes, there are multiple bounded-collusion functional encryption schemes. We present them here, and discuss their advantages and disadvantages.

Stateful Scheme

The simplest way to extend the one-key functional encryption schemes to support $q > 1$ functional keys is to just use multiple copies of the one-key scheme. Then, to issue functional keys, the stateful scheme outputs a functional key to one of these

schemes. However, in order to be secure for up to q functional keys, we need to make sure that each functional key is for a different one-key scheme. To do this requires the scheme to keep track of which keys it has given out. If $OneQFE$ is a one-key scheme, the stateful scheme does the following, to be secure for q functional keys.

- **Initialize:** Set $state = 0$.
- **Setup(1^κ):** Run setup for the one-key scheme q times.

$$(MSK_i, MPK_i) \leftarrow OneQFE.Setup(1^\kappa) \qquad i \in [q]$$

  Output $(MSK, MPK) = ((MSK_i)_{i\in[q]}, (MPK_i)_{i\in[q]})$.

- **KeyGen(MSK, C):** Let $FK = (state, OneQFE.KeyGen(MSK_{state}, C))$. Then, set $state = state + 1$, and output FK.
- **Encrypt(MPK, x):** Output $CT = (OneQFE.Encrypt(MPK_i, x))_{i\in[q]}$.
- **Decrypt(FK, CT):** If $FK = state, k$, output $OneQFE.Decrypt(k, CT_{state})$.

Note that we have a new method Initialize which sets the state to 0, and each time a key is given out, the state increases. This also means that after q keys are given out, it will not be possible to give any more.

Here, the master secret key, master public key, and ciphertext are q times as large as their one-key counterparts, while the functional key's size is only increased additively by $\log q$. We also see that the Setup and Encrypt algorithms also take q times as long, while the other two are not significantly affected.

Compared to the GVW scheme, the ciphertexts are much smaller. In the GVW scheme, $N = O(D^2q^4)$ ciphertexts are needed to compute a depth D circuit. This is a huge saving, especially as q grows (see Section 5.2.1 for more discussion of choice of N). This however does require the key authority to keep state, in contrast with the GVW protocol.

Gorbunov-Vaikuntanathan-Wee (GVW) Scheme

Gorbunov et al. in 2012 made a bounded-collusion functional encryption scheme, using a one-key functional encryption scheme as a black box, together with Shamir secret sharing, and optionally a pseudorandom number generator [32]. This scheme depends on a few parameters, N, t, v, and S. We discuss the choice of these parameters in Section 5.2.1.

The GVW scheme supports computing polynomials up to degree D, for a fixed D, over a finite field $\mathbb{F}$. The idea behind the GVW scheme is to use the BGW multiparty computation scheme to encode inputs x_i as t degree polynomials p_i over $\mathbb{F}$, and then to calculate $C(\{x_i\})$. We note that $C(\{p_i(y)\})$ is now a degree tD polynomial over $\mathbb{F}$. Now, if we have $tD+1$ values for this, we can use polynomial interpolation to recover the value of $C(\{x_i\})$. This is exploiting the fact that in the BGW scheme, polynomials of degree at most $\log N$, where there are N parties, can be calculated without any interaction.

The GVW scheme, instead of computing C, calculates a closely related circuit $G_{C,\Delta}$, for $Z_i \in \mathbb{F}$, where,

$$G_{C,\Delta}(x, Z_1, \ldots, Z_S) = C(x) + \Sigma_{i\in\Delta} Z_i.$$

Here, the Z_i serve to add extra randomness to the output. This is needed to achieve simulation security (either adaptive or non-adaptive). Now, given a one-key FE scheme $OneQFE$, the GVW scheme is as below.

- **Setup(1^κ):** Run the OneQFE setup N times.

$$(MSK_i, MPK_i) \leftarrow OneQFE.Setup(1^\kappa) \qquad i \in [N]$$

  Output $(MSK, MPK) = ((MSK_i)_{i\in[N]}, (MPK_i)_{i\in[N]})$.

- **KeyGen(MSK, C):**

  1. Pick a set $\Gamma \xleftarrow{\$} [N]$ uniformly at random, of size $tD+1$.

2. Pick a set $\Delta \xleftarrow{\$} [S]$ uniformly at random, of size v.
3. $FK_i \leftarrow OneQFE.KeyGen(MSK_i, G_{C,\Delta}) \qquad i \in \Gamma$.
4. Output $(\Gamma, \Delta, (FK_i)_{i\in\Gamma})$.

- **Encrypt(MPK, x):** If $x = (x_1, x_2, \ldots, x_l)$,
  1. For $i \in 1, \ldots, l$, pick a random t-degree polynomial μ_i over $\mathbb{F}$ with constant term x_i.
  2. For $i \in 1, \ldots, S$, pick a random tD-degree polynomial ζ_i over $\mathbb{F}$ with constant term 0.
  3. $CT_i \leftarrow OneQFE.Encrypt\left(MPK_i, (\mu_1(i), \ldots, \mu_l(i), \zeta_1(i), \ldots, \zeta_S(i))\right)$.
  4. Output $(CT_i)_{i\in\{1,\ldots,N\}}$.

- **Decrypt(FK, CT):** If $FK = (\Gamma, \Delta, \{k_i\}_{i\in\Gamma})$, use polynomial interpolation to find the unique polynomial p such that $p(i) = OneQFE.Decrypt(k_i, CT_i)$ for all $i \in \Gamma$, where p has degree at most tD. Output $p(0)$.

Where the above algorithm chose Γ, Δ uniformly at random, using a pseudorandom number generator instead ensures that a function receives the same functional key if queried multiple times. Additionally, while this scheme above only works for bounded degree polynomials, it can be bootstrapped to any polynomial sized circuit via randomized encodings [40, 4]. As we do not use them here, we leave the details to the paper by Gorbunov et al. [32].

The GVW scheme has the advantage that, unlike the stateful scheme, it does not need to keep state. This means that the key authority both doesn't need to keep track of keys being issued, but also admits the possibility that there could be multiple parties issuing functional keys, and without being required to keep a synchronized shared state, other than the shared master secret key. It also has another advantage over the stateful scheme. The stateful scheme is limited to issuing q functional keys to achieve its security. If it issues more than q keys, there is a significant chance $(\frac{1}{q})$ that security is broken, depending on which keys are shared. The GVW scheme can

issue more than q keys, and still be secure against a 2-key collusion. Rather, the GVW scheme needs q keys collaborating to break security. As long as the parties are relatively unlikely to collude, the GVW scheme can issue more than q keys and still remain highly secure as long as one party does not obtain q keys.

The GVW scheme has the disadvantage, as we will discuss in Section 5.2.1, that the size of ciphertexts, master public key, and master secret key grow with q^4 and D^2, and the functional key with q^2. If using bootstrapping, the value of D is bounded, and so the D^2 term disappears.

Agrawal-Rosen Online-Offline Functional Encryption

In a recent work, Agrawal and Rosen defined online-offline functional encryption [3], where most of the work is done offline ahead of time, and the encrypt stage takes linear time in the message size. Additionally, the total ciphertext sizes (including the offline stage) grows with $q^2|C|$, where $|C|$ is the size of the circuit to be computed. This is asymptotically significantly better than the GVW scheme using the SS scheme as the base one-key FE scheme, which grows as $q^4|C|$. This scheme exploits similarities between the dual Regev Ring Learning with Errors cryptosystem [57, 29], and symmetric key ciphertexts in the Brakerski-Vaikuntanathan fully homomorphic encryption cryptosystem [15]. They use this to first support linear functions. They then add more information that both allows multiplication to be decrypted, and also restricts what can be recovered from the fully homomorphic encryption scheme.

This scheme does have a couple disadvantages. First, compared to the GVW scheme, it makes slightly stronger, if still well accepted, assumptions about the hardness of lattice problems. Second, implementations of fully homomorphic encryption often have large constant factors [28, 37]. This scheme also does not use a one-key functional encryption scheme as a black box, which can be thought of as an advantage, in that redundancy can be removed, but also a disadvantage in that it does not gain advantages of better one-key functional encryption schemes. For example, the GVW scheme, using the GKPVZ scheme as a black box, still has ciphertexts growing with q^4, but it no longer grows with $|C|$.

Chapter 4

A Framework for Investigating Functional Encryption (FIFE)

We built the Framework for Investigating Functional Encryption (FIFE), in order to address the dearth of information on the practicality of functional encryption. FIFE allows for measuring the performance of different functional encryption schemes and evaluating their dependence on different building blocks. We aimed to make it easy to compare different functional encryption schemes, and determine how theoretical efficiency translates into practice.

Choice of Language FIFE [70] was written in C++. This language was chosen for a few reasons. First, the SS scheme given in Section 3.3.1 heavily uses Yao's garbled circuits, and we wanted to use the existing library libgarble [36], a C library improving on the library JustGarble [7], to do so. Next, the Palisade library is in C++, and supports many tools and primitives for lattice-based cryptography [60]. While we did not use it, many of the future directions for this work, discussed in Chapter 6, involve lattice-based cryptography, for which Palisade is a good fit. Finally, as a library designed to measure performance of various functional encryption schemes, C++ offered us many useful features, such as templates, while still giving us fast performance.

Existing Libraries Utilized Our library relies on existing libraries for specialized tasks, so that we did not have to implement everything from scratch. We summarize our external dependencies here, and discuss each one in more detail below.

We used *libgarble* for constructing and evaluating garbled circuits [36]. We used *crypto++* for public-key and secret-key encryption schemes, including RSA and AES [18]. We used *msgpack* to write and read keys and ciphertexts from file [25]. Finally, we used *NTL* to handle polynomials and do polynomial interpolation [69].

Library Structure FIFE was constructed in a modular fashion, so that we could additionally make a detailed study of the different choices of building blocks and parameters for specific functional encryptions schemes.

Our library has four major parts. Each part corresponds to a major choice that can be made about functional encryption implementations. The first part corresponds to a choice of traditional encryption scheme. The next part is for what class of circuits to use. Then, there are different one-key FE schemes to use. Finally, there are bounded-collusion functional encryption schemes. While we have not included multiple implementations for each of these choices, our implementation is structured in a way to allow additional options to be easily added. In addition to these parts, our library also heavily uses garbled circuits. Additionally, we added the ability to write the keys and ciphertexts to file, to allow them to be communicated. Also, we wrote a test suite to ensure the correctness of our constructions. We describe each of these in more detail below.

4.1 Traditional Encryption Schemes

Our library offers the user a choice of traditional encryption schemes, including both public-key encryption and secret-key encryption. We chose to present a uniform interface for all such schemes, so that they can be used in a black box fashion. We did this in the following way.

First, we created a base class for traditional encryption schemes. The templated

class PKEBase has the default interface, which takes as a parameter a description of the PublicKey, SecretKey, and CipherText types for a specific encryption scheme. For a secret-key encryption scheme, the PublicKey and SecretKey are the same, but this allows us to create a more general interface for any traditional encryption scheme.

This class then uses these types to specify the Setup, Encrypt, and Decrypt functions, as described in Section 3.2.2, where the plaintext is always a byte array across encryption schemes. This design allows us to use one interface for all traditional encryption schemes, without restricting the types of keys or ciphertexts.

This design also allows the use of preexisting implementations of these traditional encryption schemes. The only thing required is to create a wrapper which fits this interface for each encryption scheme chosen. We chose to do so for a public-key encryption scheme, RSA, and a secret-key encryption scheme, AES. We chose these schemes for their popularity and prevalence. We started with the library crypto++, a standard library used for cryptography in C++ [18].

For RSA, we chose to use the RSA_OAEP_SHA option, which is RSA with Optimal Asymmetric Encryption Padding, using SHA [8]. We made this choice as it has been proven to be secure, assuming RSA is hard [24]. For AES, we chose to use AES in CFB Mode, or cipher feedback mode. We chose this mode as it does not require the message to be padded.

4.2 Circuits and Universal Circuits

An important part of our scheme is the circuit families it supports. We started by supporting three classes of circuits. First is parity circuits. A parity circuit computes the parity of some subset of the input bits. This class is equivalently inner products between the message and a vector in $\mathbb{Z}_2$.

Next, we supported inner product circuits modulo and prime p. This is a generalization of the above parity circuits. These circuits also work well for the GVW scheme, as that scheme computes polynomials over a finite field, and $\mathbb{Z}_p$ is a natural finite field, with inner product as a simple polynomial to evaluate.

Circuit Type	SS Scheme	Stateful Scheme	GVW Scheme
Hamming Distance	yes	yes	no
Inner Product mod p	yes	yes	yes
Parity	yes	yes	no

Table 4.1: The types of circuits implemented, and which schemes support them.

We also support computing Hamming distance. These circuits give the Hamming distance between the message and a fixed string specified by the circuit.

Of these three, we didn't support the first and third for the GVW scheme, as they do not compute over a finite field large enough. The circuit types we implemented and the schemes which support each are summarized in Table 4.1.

We made universal circuits for each of these classes of circuits. However, we did not implement the general universal circuits we discussed in Section 3.2.4. These circuits were sufficient to do an initial study into the performance of the functional encryption schemes, while still providing interesting functionalities. Also, given our results in Sections 5.1 and 5.2, we saw that the size of the ciphertexts was already too large for large circuits, without considering using a universal circuit.

To build a universal circuit for a class of circuits, we broke it down into standard components, both for ease of building, and for ease of verifying correctness. These utility circuits also allow each component to be separately tested. For each of the components discussed below, we speak of them taking in some inputs, and having an output. What we mean by this is that given some wires in a circuit being built, the component adds on to that circuit a subcircuit which applies the specific operation to the input wires, with the result of that operation going on the output wires.

ZeroOne Gate The ZeroOne gate takes in two bits, and outputs 1 if the first bit is 0 and the second is 1. This was added for convenience, as it is not a standard gate supported by libgarble. We implemented this with an XOR gate and an AND gate. We could have instead added this gate to libgarble. We chose not to do so since, as we will discuss in Section 4.6.1, we don't write the garbled table for XOR gates to file, it does not increase the size of our ciphertexts by much. This also allows us to use the libgarble library as is.

Mux The mux takes two inputs of n bits and a decider bit, and outputs one of the inputs based on the decider bit. We implemented this as n single-bit muxes.

Add The adder takes two n-bit integers in, and outputs an $(n+1)$-bit integer of their sum. We implemented this as a half-adder, which adds two bits and outputs their sum and carry bit, followed by $n-1$ full-adders, which add two bits, along with the carry from the previous addition, and outputs the result and a new carry bit. Additionally, we used the design by Kolesnikov, Sadeghi, and Schneider, which is optimized for garbled circuit efficiency [47].

Subtract This component takes in two numbers, and outputs the first minus the second. It also outputs a bit corresponding to the sign of the result. This is structurally very similar to the adder, but has some modifications to subtract instead of add. This design was also due to Kolesnikov et al. [47].

Reduce mod p This component takes a number between 0 and $2p$, for some p, and outputs a number equivalent modulo p. This is implemented as subtracting p, checking the sign of the result, and using a mux to choose based on that the original or the new result.

Add mod p This component takes two numbers and returns their sum reduced mod p. We do this by adding the numbers, then reducing mod p, with our above described components. It assumes that the inputs are already reduced mod p.

Double mod p This component takes a number, and returns twice that number reduced mod p. We do this by adding a wire fixed to 0 at the end of the input, then reducing it mod p. After this reduction, we now know the highest order bit is a 0, so we can ignore it.

Multiply mod p This component takes two numbers less than p, and outputs their product reduced mod p. We implement this as follows. Here, we let $n = \lfloor \log p + 1 \rfloor$,

the number of bits in the numbers. Given two numbers, a and b, we go through the bits of b from highest-order to lowest-order, and accumulate the correct number of copies of a. We use our above double mod p, add mod p, and mux components to do the following.

1. Let $sum = 0$, and $bit = n - 1$
2. While $bit \geq 0$,
   (a) $sum = (sum * 2) \mod p$.
   (b) If $b[bit] = 1$, let $sum = (sum + a) \mod p$.
   (c) Let $bit = bit - 1$.
3. Output sum.

Add in $GF(2^n)$ This component takes two elements of $GF(2^n)$, and returns their sum. Here, we use an n-bit representation for elements of $GF(2^n)$, corresponding to the x^i terms of $\mathbb{Z}_2[x]/p(x)$, for an irriducible degree-n polynomial $p(x)$. This sum is just the XOR of the corresponding bits of the two elements. Note that we do not need the polynomial $p(x)$ to perform addition in $GF(2^n)$. Also, note that since addition and subtraction are equivalent in $GF(2^n)$, this is also subtraction in $GF(2^n)$.

Reduce in $GF(2^n)$ This component takes an element of $GF(2^n)$, and a bit corresponding to whether it needs to be reduced, and outputs the element which is reduced if necessary. This also requires as input the irriducible polynomial for the representation of $GF(2^n)$. This is implemented as an XOR gate and a mux for each term in the irriducible polynomial. There will usually be 2 or 4 such terms [64].

Multiply in $GF(2^n)$ This component takes two elements of $GF(2^n)$, and returns their product. This also requires as input the irriducible polynomial for the representation of $GF(2^n)$. We implemented this similarly to multiplication mod p, except instead of subtracting p to reduce, we add the irriducible polynomial. Also, instead

of checking whether a number is greater than p, we only need to check if the product has a degree n term, which requires fewer gates.

Add mod 2^{32} This component takes two 32-bit numbers, and adds them mod 2^{32}. It is implemented as adding the numbers using the previous component, and ignoring the highest order bit of the sum.

Multiply mod 2^{32} This component takes two 32-bit numbers, and multiplies them mod 2^{32}. This is done as in multiplying mod p, but we do not reduce mod p, and instead ignore terms of 2^i for $i \geq 32$.

Hamming distance This component takes two inputs with n bits, and outputs the Hamming distance between them. This is implemented as first XORing the corresponding bits together, then adding the number of differences in a tree-like fashion, adding neighboring numbers together until we have one sum left. This allows us to add numbers with fewer bits lower in the tree, to save gates used in the circuit. This circuit for counting number is similar to the parallelized counter circuit of Huang et al. [39].

The complexity of each component is described in Table 4.2. We used these utilities to implement our universal circuits for the circuit families in Table 4.1. For parity circuits, we first had AND gates between the message and our circuit description, and then took the XOR of all of those outputs. For inner product circuits, we multiplied each pair of elements from the message and circuit description mod p, and then added them all up mod p, using our utilities for those. Finally, for Hamming distance, we used our utility for Hamming distance. The sizes of these circuits is given in Table 4.3.

Component	Inputs	Total Gates	non-XOR gates
ZeroOne	(1,1)	2	1
Mux	(n,n,1)	$4n$	$2n$
Add	(n,n)	$5n-3$	n
Subtract	(n,n)	$6n$	n
Reduce mod p	n	$10n-4$	$3n-2$
Add mod p	(n,n)	$15n+3$	$4n+1$
Double mod p	n	$10n+6$	$3n+1$
Multiply mod p	(n,n)	$29n^2+9n$	$9n^2+2n$
Add in $GF(2^n)$	(n,n)	n	0
Reduce in $GF(2^n)$	(n,c†)	c	0
Multiply in $GF(2^n)$	(n,n,c†)	$5n^2+nc$	$2n^2$
Add in 2^{32}	(32,32)	157	32
Multiply in 2^{32}	(32,32)	9120	3072
Hamming Distance	(n,n)	$8n^\ddagger$	$2n^\ddagger$

Table 4.2: Number of gates and non-free gates for each circuit component, in terms of their input length(s). †c is the number of terms in the irriducible polynomial, and is usually 2 or 4 [64]. ‡The gates for Hamming distance are not exact, and instead upper bounds.

Component	Inputs	Total Gates	non-XOR gates
Hamming Distance	n bits	$8n^\dagger$	$2n^\dagger$
Parity	n bits	$2n$	n
Inner Product mod p	e n-bits elements	$e(29n^2+24n+3)$	$e(9n^2+6n+1)$

Table 4.3: Number of gates and non-free gates for each circuit component, in terms of their input length(s). †The gates for Hamming distance are not exact, and instead upper bounds.

4.3 One-key Functional Encryption

4.3.1 Singleton FE Scheme

The singleton FE scheme is the FE scheme that encrypts only one circuit, the identity circuit, where $C(x) = x$ as defined by Gorbunov et al. [32]. The implementation of this scheme was straightforward, and just follows the description in Section 3.3.1. We used the pair data structure from the C++ standard library to hold the pairs of keys and ciphertexts, and created a struct to hold the bit and key from KeyGen.

4.3.2 Sahai-Seyaglioglu Scheme

We implemented the SS scheme for one-key functional encryption, as described in Section 3.3.1. We started out by initializing the scheme for a specific class of circuits. Then, the four algorithms were implemented as follows.

- **Setup(1^κ):** The Setup algorithm implementation is straightforward from the description, and creates a vector of pairs of public keys and secret keys as the master public key and master secret key, respectively.

- **KeyGen(MSK, C):** The KeyGen algorithm outputs functional keys, which also include a description of the circuit to be evaluated, in addition to the secret keys corresponding to the description. This means that in this implementation, it is not possible to give someone a key without them knowing what function it corresponds to. Here, the circuit is a vector of integers that are 0 or 1, and the functional key is a struct with that vector and a vector of secret keys of the base encryption scheme.

- **Encrypt(MPK, x):** The Encrypt algorithm starts by generating a universal circuit for the class of circuits, that takes x and a description of a circuit C. It then garbles this circuit to produce a garbled circuit G. Next, we take the labels corresponding to a circuit description C, and encrypt the label for bit i for b with $MPK_{i,b}$, where $b \in \{0, 1\}$. The ciphertext now consists of three

parts: G, the encrypted labels for a circuit description, and the input labels for G corresponding to x. Here, the ciphertext is a struct with a garbled circuit from libgarble, a vector of labels for x, and a vector of pairs of ciphertexts of the base encryption scheme for the encrypted labels.

- **Decrypt(FK, CT):** The Decrypt algorithm uses the functional key to decrypt the labels corresponding to the circuit description. Then, combined with the labels for from the ciphertext, it evaluates the garbled circuit, and outputs the computed result.

4.4 Bounded-Collusion Functional Encryption

4.4.1 Stateful Scheme

The implementation of our stateful scheme is just a straightforward translation of the scheme described in Section 3.3.2, where the keys and ciphertexts are vectors of the base one-key scheme, and the functional key is a struct with an integer corresponding to which scheme it corresponds to and a one-key scheme functional key.

4.4.2 GVW scheme

We implemented the GVW scheme for bounded-collusion FE. This uses the one-key scheme as a black box. When initializing the scheme, it sets up the security parameters, N, t, v, and S, as discussed in Section 3.3.2, and the one-key scheme used. See Section 5.2.1 for more information about the choice of these parameters. It also takes in the degree D of the polynomials to be computed over the inputs and the number of keys for which it should be secure, q. We used NTL to make the polynomials and to perform the polynomial interpolation [69].

We also added in the option to disable using the subsets Δ, and the polynomials ζ_i. This shrinks the size of the keys and ciphertexts, at the cost of not achieving simulation security. See Section 5.2.3 for more discussion on the space usage impact of this choice, and Section 3.3.2 for discussion on the security implications.

Additionally, we chose not to implement the bootstrap operations, as we only considered some circuits (see Table 4.1 for which ones we implemented), and these all had low degree.

- **Setup(1^κ):** The Setup algorithm is implemented in a straightforward manner from the description in Section 3.3.2, where the master public key and master secret key are vectors of the base one-key versions of each.

- **KeyGen(MSK, C):** The KeyGen algorithm first randomly picks $tD + 1$ of the one-key master secret keys that comprise the master secret key. Then, if using Δ, it picks v elements of $[S]$ randomly, as Δ, and includes it in the circuit description. It now uses the one-key KeyGen algorithm with the $tD + 1$ master secret keys to generate $tD+1$ one-key functional keys for this circuit description, and returns them. The functional key is a struct with a vector containing Γ, which says which $tD + 1$ one-key schemes the key corresponds to, a vector with Δ, which says which v values to add, and a vector of one-key functional keys.

- **Encrypt(MPK, x):** The Encrypt algorithm, on an input message $x = \{x_i\}$, for each i, picks a random degree t polynomial p_i, and sets the constant coefficient for the polynomial to x_i. Now, if using Δ, it also generates S random degree tD polynomials, ζ_i, with constant coefficient 0. Then, for each $j \in \{1, \ldots, N\}$, it uses the one-key Encrypt algorithm on the inputs $p_i(j)$, and if using Δ, $\zeta_i(j)$, and generates N one-key ciphertexts, and outputs those. The ciphertext is just a vector of one-key ciphertexts.

- **Decrypt(FK, CT):** The Decrypt algorithm, given $tD + 1$ one-key FE secret keys, uses them to call the one-key Decrypt algorithm on each of the corresponding $tD + 1$ ciphertexts. Then, it uses these pairs of j, and the circuit $C(\{p_i(j)\})$ to do polynomial interpolation to get the polynomial $C(\{p_i(x)\})$. Then, we evaluate that at 0, to get $C(\{p_i(0)\}) = C(\{x_i\}) = C(x)$. If we are using Δ, the process is exactly the same, since $\zeta_i(0) = 0$ for each i, and so the resulting polynomial still has the same value at 0.

4.5 Garbled Circuits

We decided to use libgarble to garble circuits [36]. This library is based on JustGarble [7]. However, libgarble also includes half gates [74], in addition to the existing improvements in JustGarble, like free-XOR for garbled circuits [48]. We use it in half-gates mode, which allows us to reduce the space that the garbled circuits take.

Using libgarble limits our security to that of AES128. This is the case since libgarble uses AES128 as its primary mean of encrypting and decrypting labels. Thus, any system we build using it cannot be more secure than AES128.

4.6 File Input and Output for Keys and Ciphertexts

In order to have a usable encryption scheme, we need to support setup, key generation, encryption, and decryption to be performed by separate parties. Phrased more concretely, we need to be able to write and read the master secret key, master public key, functional keys, and ciphertexts to and from file, so that they can be transferred between parties.

We write to file in two stages. First, we use msgpack to pack our objects into the msgpack format, a binary serialization format [25]. Then, we write this packed object to file. To read from file, we do it in reverse order; we first read the file into a buffer, and then use msgpack to unpack it into the original object.

The msgpack format has some primitive types, such as integers, booleans, and byte arrays, along with derived types such as arrays. It packs these by generating a code corresponding to the type of the value and information about it, followed by their value. For integers, this means that it stores them with only an extra byte of space. Booleans take exactly one byte, and binary arrays are stored as the length of the array, followed by the array. Msgpack also supports other types, but we did not use them [25]. Arrays are implemented similarly to byte arrays, where writes their length, followed by the items. However, the items in an array are not just binary, but instead more objects encoded with msgpack. This allows us to easily support

complex data structures, as long as all of its elements are supported by msgpack. The msgpack library for C++ also has convenient macros to support reading and writing many data structures, as long as the types they use already have definitions for how to write and read them. This meant that we only needed to implement a few of these, and the rest then were defined by the macros.

RSA keys The first msgpack format we defined was for writing RSA public and secret keys. The RSA keys consist of two or eight numbers of type Integer, as defined by crypto++, each of which have up to the security parameter bits. We first implement writing and reading a crypto++ defined Integer. crypto++ has functions which gives the size of the encoding of the integer in bytes, and another which encodes it to a byte array. We used these to encode the Integer as a binary array. To unpack it, we use the Decode function as defined by crypto++ for the Integer class to turn the byte array back into an Integer.

Now, since the public key can be represented by two of these Integers, we pack the public key as an array of two integers. The private key can similarly be represented as eight Integers, and so we pack them as an array of eight integers. To unpack them, we just unpack the Integers using our implementation, and use the constructors for the public and private keys in crypto++ which take two and eight Integers respectively.

AES keys We next defined how to pack and unpack an AES key. Since it is a secret key scheme, there is only one type of key. AES keys are just binary, so we pack them as byte arrays of the appropriate length. To unpack them, we similarly unpack the byte array, and construct the key with that value.

SS Ciphertext We then defined how to pack and unpack the ciphertext for the SS one query FE scheme. The ciphertext consists of a garbled circuit, a vector of the labels for the garbled circuit corresponding to the encrypted message, and a vector of the encrypted labels that will be used for the circuit description. This is then implemented as an array of size three corresponding to these parts. The last two parts are just stored as arrays of binary inputs, and are packed and unpacked as we

have previously discussed.

The garbled circuit is packed first as an array of size two, corresponding to the structure of the circuit and the cryptographic material. In the first element, we pack an array of the appropriate size containing the following. First, we pack the parameters for the garbled circuit, m, n, q, r, and fixed_wires, which are the output size in bits, input size in bits, number of gates, number of wires, and number of wires that are fixed to either 0 or 1. We then pack the q gates as the two input wires, the output wire, and the type of the gate. Finally, we pack the outputs. This part of the packing is based on how libgarble packs a garbled circuit using msgpack [36].

The second element of the array, with the cryptographic material, is packed as a single binary block, which contains the garbled tables for the gates of the garbled circuit, the fixed label used for wires that are fixed to either 0 or 1, and the global key used for AES by libgarble.

4.6.1 Space Saving

As we will discuss in Section 5.1.5, the above packed ciphertexts, and specifically the garbled circuit, are very large from a space standpoint. To try and alleviate this, we tried to minimize the size of this ciphertext. As libgarble uses half gates, for each gate we only need to store two values in the garbled table. Since it also uses free XOR, XOR gates do not use the garbled table, and so we also do not write to file the garbled table for XOR gates. Depending on the number of XOR gates in the circuit, this can cut the space usage significantly. We also noticed that we could make similar savings for NOT gates by implementing them as an XOR gate with the fixed zero wire.

We also investigated the possibility of reusing input labels for the GVW bounded-collusion FE scheme. The idea is that since this ciphertext consists of many copies of the one key ciphertext, and to decrypt it we would plug the same function into all of them, we can use the same labels across all these ciphertexts for the circuit description. However, as we will discuss in Section 5.1.5, we determined that this was a very small saving. Additionally, implementing it would mean having a non-modular

implementation of writing the ciphertexts for the GVW scheme to file, and we decided it was not worth that cost.

4.7 Testing Framework

To make sure our code was correct, we used Google Test, or gtest, to make a test suite [21]. We wrote a few classes of tests. First, we wrote tests to ensure that our various circuit utilities correctly computed the circuits they were supposed to build when garbled. For each test, we created a new garbled circuit and initialized it, ran the circuit utility with the appropriate inputs, and then finished building the garbled circuit, garbled, and evaluated it. We then checked that the value returned was the value we expected to get. We also used this to ensure that each implemented FE scheme worked correctly.

We also wrote tests to ensure that writing and reading keys from files was working correctly. For each object that we support writing to file, we tested this by comparing the object that was written to file and read against the one that stayed in memory. For the traditional encryption schemes, we generated keys, wrote them to file and read them from file, and ensured that these keys worked with the keys that were in memory, and gave the same results. For the FE schemes, we did the same thing, but for each of the master secret key, master public key, functional key, and ciphertext.

Chapter 5

Results

Using the framework we built, we investigated the performance of different bounded-collusion functional encryption schemes and their components. First, we looked at the Sahai-Seyalioglu (SS) scheme (discussed in Section 3.3.1). We looked at how its performance depended on the base encryption scheme it used. We also looked at its performance for different families of circuits and for different sized circuits.

We next looked at the Gorbunov-Vaikuntanathan-Wee (GVW) scheme (discussed in Section 3.3.2). We first investigated how the choice of parameters for the GVW scheme affected its security, and found what values corresponded to different security levels. We then looked at its performance at these security levels for different bounds on the number of collusions for which it was secure and depths of the circuits to be evaluated. We then looked at the cost of providing adaptive simulation security.

We also compared the GVW scheme to the stateful scheme. We looked at instances of the stateful scheme that support the same number of keys as the GVW scheme, and also at instances with similar key and ciphertext sizes, and looked at the difference in the number of keys supported.

We finally compared our implementations to those of Controlled Functional Encryption [54].

Experimental Setup In each of our experiments, we measured the sizes of the master secret key, master public key, functional key, and ciphertext. In addition,

Scheme	Setup	KeyGen	Encrypt	Decrypt
AES128	1.928	0.064	4.927	1.008
AES256	2.147	0.076	5.770	1.048
RSA1024	2396.458	0.648	13.401	46.043
RSA2048	15463.690	0.734	15.835	203.116
RSA4096	184937.500	0.953	30.158	1292.402

Table 5.1: Running times for the SS scheme with inner product circuits mod 8123, for varying base encryption schemes. All times are in milliseconds.

we also measured the time that each of the Setup, KeyGen, Encrypt, and Decrypt algorithms took. All of our experiments ran on a Lenovo Y510p computer with an Intel Core i7-4700MQ (2.40GHz) processor, 8GB memory, and Ubuntu 14.04. Additionally, all times reported are for the average of 10 runs.

5.1 Performance of the Sahai-Seyalioglu Scheme

We investigated the performance of the Sahai-Seyalioglu scheme in detail, both for its own sake, and since all of our bounded-collusion functional encryption schemes use it as a black box. We looked at the impact of using different base encryption schemes, and the performance for each type of circuit we study, namely parity circuits, inner product circuits modulo a prime, and Hamming distance.

We observe that this scheme is generally reasonably fast. However, the ciphertext sizes grow large with complex circuits. The biggest bottleneck in extending this scheme to more complicated functions is the size of the circuits being evaluated.

5.1.1 Different Base Encryption Schemes

We look at the impact of using different base encryption schemes for the SS scheme. We compare these for inner product circuits mod 8123, a 13-bit prime, for length 10 vectors. We consider AES with 128 and 256 bit keys, and RSA with 1024, 2048, and 4096 bit keys. We consider this for the SS scheme, as all other schemes we looked at use one-key functional encryption schemes in a black-box manner. The running times are in Table 5.1, and the space usages are in Table 5.2.

	MSK		MPK		FK	CT
Scheme	size	keys	size	keys	size	size
AES128	4,814	260	4,814	260	2,477	1,036,937
AES256	8,974	260	8,974	260	4,557	1,036,937
RSA1024	154,568	260	34,974	260	77,356	1,061,377
RSA2048	304,838	260	68,514	260	152,489	1,094,917
RSA4096	605,669	260	135,074	260	302,906	1,161,477

Table 5.2: Space usages for keys and ciphertexts for inner product circuits mod 8123 with length 10 vectors using the SS scheme, for varying base encryption schemes. Sizes are in bytes, and keys are the number of base keys the object is composed of.

From the data in Table 5.1, we can see that RSA in general takes much longer than AES, especially in the setup phase, where it is many orders of magnitude slower. However, the decrypt phase is also significantly slower, especially for large RSA key sizes. One way to continue to use RSA is to generate the keys offline, and read them from memory when needed. We generated 80000 RSA3072 key pairs offline, in 32,275.7 seconds, or in just under 9 hours.

From Table 5.2, we also see that RSA keys are larger than their AES counterparts. However, this difference is not as important in the ciphertext, since inner product is a relatively complex circuit compared to, for example, hamming distance. Indeed, as Section 5.1.5 discusses, when using AES, the ciphertexts are a negligible fraction of the total ciphertext size. We see here that using RSA can increase this to a 10% difference, which would be more acutely felt when computing hamming distance. For AES we note that AES256 does not change the size of the ciphertext, only the keys.

Singleton Functional Encryption Using the Singleton FE scheme as the base encryption scheme gives the Sahai-Seyalioglu scheme adaptive simulation security. We now look at the impact of this increased security on performance. By design, it will double the master secret key and master public key sizes, and add only one bit for each key to functional keys, so we only consider the impact on ciphertexts, which changes based on the size of the circuit, ranging from a negligible increase to doubling the size of the ciphertext. We again consider this for the SS scheme. Table 5.3 shows the increase in ciphertext sizes for each of the encryption schemes considered.

Scheme	Base Ciphertext Size	Singleton Ciphertext Size	Ratio
AES128	1,036,937	1,047,077	1.009
AES256	1,036,937	1,047,077	1.009
RSA1024	1,061,377	1,095,957	1.032
RSA2048	1,094,917	1,163,037	1.062
RSA4096	1,161,477	1,296,157	1.115

Table 5.3: Space ciphertexts for inner product circuits mod 8123 with length 10 vectors using the SS scheme, for varying base encryption schemes. Sizes are in bytes, comparing between using the Singleton version of a scheme and the scheme itself. Ratio is the ratio of the Singleton ciphertext to the base ciphertext

Length	Setup	KeyGen	Encrypt	Decrypt
1	0.119	0.011	0.092	0.024
10	0.280	0.015	0.319	0.066
100	1.831	0.064	2.492	0.456
1000	15.884	0.392	16.712	2.987
10000	123.819	4.172	168.315	29.911
100000	1232.034	41.245	1661.843	297.880

Table 5.4: Running times for the SS scheme for parity circuits of the given lengths, in bits. Times are in milliseconds.

Here we see that using Singleton FE as a base encryption scheme for inner product circuits does not increase the size of the ciphertexts significantly when using AES, and even for RSA, the increase is not too large.

5.1.2 Parity

We first studied the SS scheme for parity circuits. This is a very simple circuit and is useful as a lower bound on circuit complexity. The running times for different length parity circuits are in Table 5.4, and the key and ciphertext sizes are in Table 5.5.

From these results, we see what we expect; all of the times and sizes increasing approximately linearly with the length. This also gives us lower bounds on what we should expect for running times and space usages for other circuits.

	MSK/MPK		FK	CT
Length	size	keys	size	size
1	39	2	22	177
10	372	20	193	1,352
100	3,704	200	1,907	13,822
1000	37,004	2,000	19,007	141,677
10000	370,004	20,000	190,007	1,419,683
100000	3,700,006	200,000	1,90,0011	15,268,621

Table 5.5: Key and ciphertext sizes for the SS scheme for parity circuits of the given lengths, in bits. Sizes are in bytes.

5.1.3 Inner Product Modulo a Prime

We now investigate the SS scheme for a more useful generalization of parity circuits, namely inner product circuits modulo a prime.

Inner Product Modulo Different Primes

First, we looked at the SS one-key functional encryption scheme for inner product mod p, where p was different primes of bit lengths from 8 to 31, for a vector of length 10, using AES as our base encryption scheme. The times each algorithm took for different length primes is presented in Figure 5-1, and the space that the keys and ciphertexts took is presented in Figure 5-2.

From these graphs, we first see that the KeyGen algorithm is significantly faster than the others, to the extent that it appears to take no time. We also see that the Encrypt algorithm takes the longest, and appears to be growing superlinearly, but is still only a small factor slower than the KeyGen and Decrypt algorithms for these lengths. We also see that the ciphertexts grow superlinearly with the length of the prime, which matches our expectation, given the number of gates we calculated in Table 4.3 grows quadratically with the length of the prime. These graphs also show that the ciphertexts are multiple orders of magnitude larger than the keys.

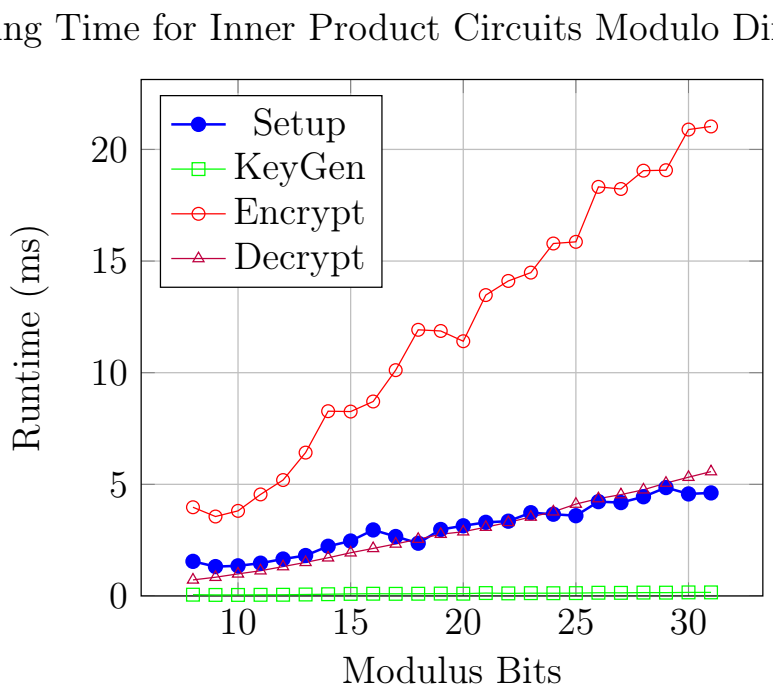

Figure 5-1: SS running times for different primes, with AES, inner product mod p circuits, for length 10 vectors. The data used to generate this figure is in Table C.1.

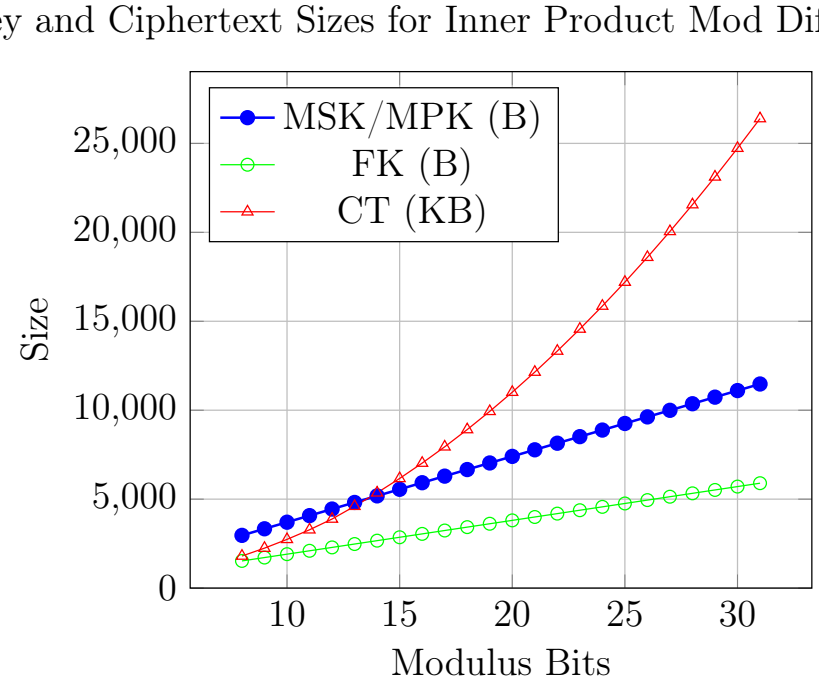

Figure 5-2: SS key and ciphertext sizes for different primes, with AES, inner product mod p circuits, for length 10 vectors. The data used to generate this figure is in Table C.2.

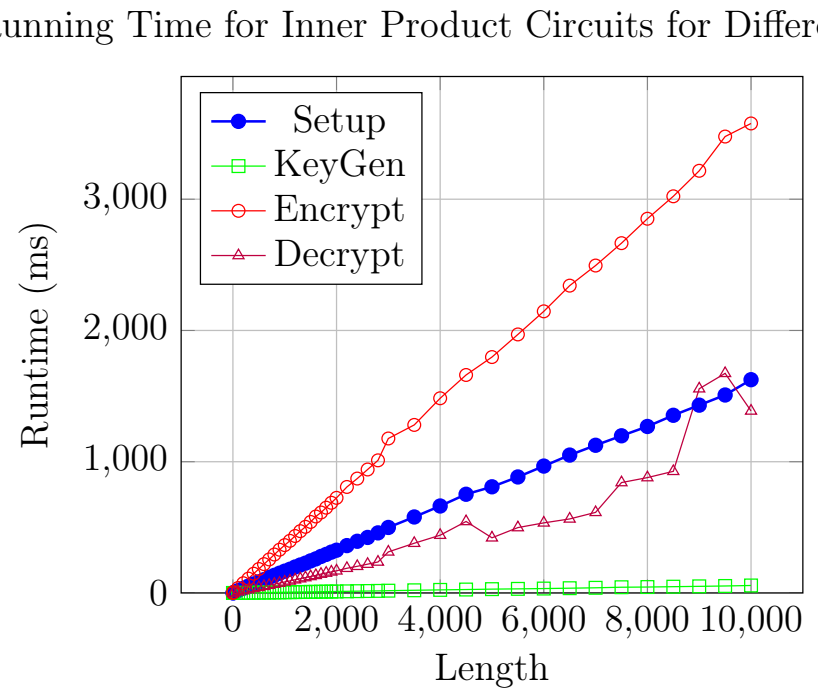

Figure 5-3: SS running times for different lengths, with AES, inner product mod 8123 circuits. The data used to generate this figure is in Table C.3.

Inner Product Modulo p, for Different Vector Lengths

We also looked at the SS one-key functional encryption scheme for different length vectors for inner product mod 8123, using AES as our base encryption scheme. The times each algorithm took for different lengths is presented in Figure 5-3, and the space that the keys and ciphertexts took is presented in Figure **??**.

The running time of the algorithms is in line with what we would expect. The running time of the Decrypt algorithm is less smooth than the other ones, as the test computer was close to running out of memory on those runs. We also see that the ciphertexts grow linearly with the length of the input vector, to approximately 1.3 Gigabytes for a length 10000 vector, or about 130 Kilobytes per element of the vector. This is a fairly large limiting factor on the size of vectors that can be supported by a functional encryption scheme.

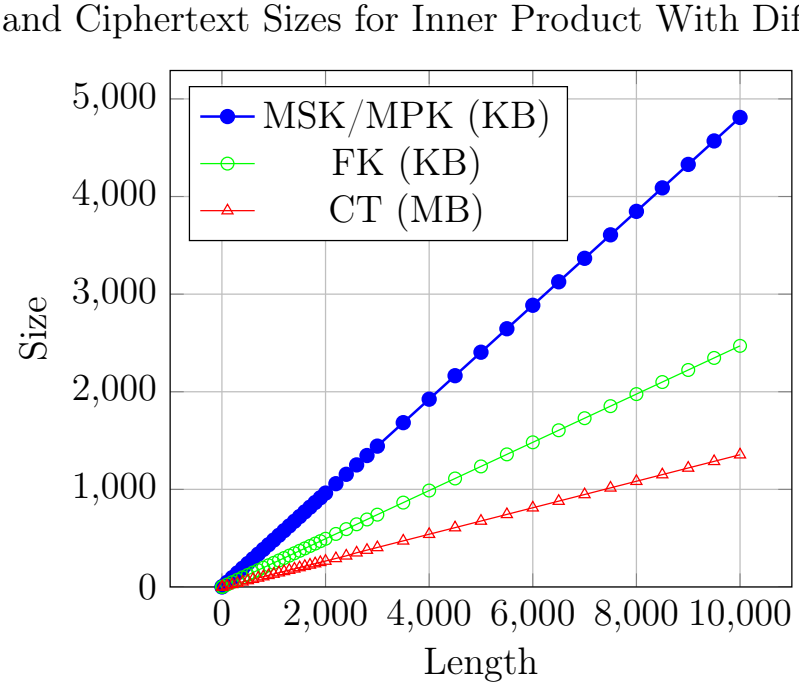

Figure 5-4: SS key and ciphertext sizes for different primes, with AES, inner product mod p circuits, for length 10 vectors. The data used to generate this figure is in Table C.2.

5.1.4 Hamming Distance

We also tested circuits for Hamming distance in the SS Scheme for different length input vectors. We chose to look at Hamming distance because it is a simple type of circuit, that nevertheless is still used in error correcting codes, among other things. Additionally, it is one of the common circuits implemented to test the efficiency of a scheme. The running times for various length inputs are in Table 5.6, and the space usages are in Table 5.7.

Bits	Setup	KeyGen	Encrypt	Decrypt
10,000	127.448	4.081	168.927	30.422
16,000	197.898	6.518	265.079	48.095
20,000	249.972	8.323	342.458	60.525
60,000	748.012	24.875	1005.205	178.330
1,500,000	18421.860	585.208	25109.850	4476.482

Table 5.6: Running times for SS scheme for Hamming distance for different length inputs, in milliseconds.

Bits	MSK size	MSK keys	FK size	CT size	Circuit Gates
10,000	370,004	20,000	190,007	2,520,831	79,953
16,000	592,004	32,000	304,007	4,245,911	127,933
20,000	740,004	40,000	380,007	5,405,371	159,948
60,000	2,220,004	120,000	1,140,007	16,787,303	479,938
1,500,000	55,500,006	3,000,000	28,500,011	422,866,997	11,999,923

Table 5.7: Key and Ciphertext Sizes for SS scheme for Hamming distance for different length inputs in bytes.

5.1.5 Analysis of Ciphertext Components

From the above results, we can see that the ciphertexts sizes are the largest practical barrier for the SS scheme. In comparison, the running times were more reasonable, with the longest at approximately 25 seconds for computing Hamming distance for 1,500,000 bit vectors. Given this, it is useful to break down the parts of the ciphertext for the SS scheme by size.

As we described in Section 4.3.2, the ciphertext in our SS implementation is composed of a vector of input labels for the message, a vector of pairs of encrypted labels for the circuit, and the garbled circuit. For complex circuits, the garbled circuit can account for more than 98% of the space usage. For some circuits, we present the sizes of the ciphertexts, and the fraction which is taken by the garbled circuit, in Table 5.8.

These compositions suggests that, for simple circuits, because of the large number of input labels, encrypted or otherwise, not too much can be gained in the SS scheme by reducing the size of the circuits. Conversely, for even moderately complicated circuits like inner product modulo a prime, reducing the size of the circuit almost directly reduces the size of the ciphertexts.

Circuit	Ciphertext	Garbled Circuit Fraction
Hamming 10 bits	1,722	0.535
Hamming 100 bits	22,529	0.644
Hamming 1000 bits	232,923	0.656
Hamming 10000 bits	2,520,831	0.682
Inner Product mod 131, length 10	404,216	0.984
Inner Product mod 131, length 100	4,878,807	0.986
Inner Product mod 131, length 1000	52,026,571	0.987
Inner Product mod 131, length 10000	533,256,823	0.987
Parity 10 bits	1,352	0.408
Parity 100 bits	13,822	0.421
Parity 1000 bits	141,677	0.435
Parity 10000 bits	1,419,683	0.436

Table 5.8: Ciphertext sizes, and the fraction of that taken by the garbled circuit, for different circuits.

5.2 Performance of the Gorbunov-Vaikuntanathan-Wee Scheme

We now study the performance of the GVW scheme. First, we investigated the correct values for the parameters described in Section 3.3.2. We then studied how the GVW scheme scales with both the depth of circuit being computed, and the level of security offered. We then looked at the cost of guaranteeing adaptive simulation security.

We found that our implementation of GVW scheme is infeasible for most choices of bounds on collusion, especially at an 80-bit level of security. This is primarily due to the size of the ciphertexts. For a circuit like inner product, that can do arithmetic over a field, most circuits will be too complex to be practical, and will generate very large ciphertexts.

5.2.1 Security Parameters for GVW

The security of the GVW scheme depends on a few parameters, as discussed in Section 3.3.2: N, the number of one-key FE schemes used, t, the degree of the polynomial to represent each input, S, the number of elements added to the result of the circuit, and v, the number of random polynomials chosen per functional key. If we let the

degree of the circuit we want to compute on the inputs be D, and the number of keys under which the scheme is secure be q, Gorbunov et al. showed that with $N = \Theta(\kappa D^2 q^4)$, $t = \Theta(\kappa q^2)$, $v = \Theta(\kappa)$ and $S = \Theta(\kappa q^2)$, the GVW scheme is as secure as the one-key scheme, except with a $2^{-\Theta(\kappa)}$ probability [32]. While this gives us an asymptotic understanding of security, it tells us nothing about what the constants are in the choice of parameters. Finding these constants is necessary to instantiate these schemes in practice. In order to do this, we wrote a set of scripts in python to estimate probabilities of a security failure.

Small-Intersection Sets

In the GVW scheme, discussed in Section 3.3.2, a functional key is composed of functional keys for $tD+1$ one-key FE schemes, out of the N total one-key FE schemes. The security of the GVW scheme depends on the functional keys not sharing many one-key FE functional keys. If two bounded-collusion FE functional keys allow you to decrypt the same one-key ciphertext, then security isn't guaranteed (and in libgarble, completely fails). Since the inputs are hidden using a degree-t polynomial, the GVW scheme is as secure as the one-key scheme as long as at most t inputs are revealed.

We can rephrase this problem without referring to cryptography. Given a set of N elements, we want to select q subsets of size $tD+1$, such that the number of elements that are in more than 1 of the subsets is at most t. This is the problem that Gorbunov et al. described as the small-intersection sets problem.

We used a python script to estimate the probability that our sets do not have small intersection. We did this in a recursive manner, by estimating, for each q, the probability that the number of intersections was greater than a threshold $thresh$, over q, p_q. Our base case was $q = 2$. Here, we calculated the explicit probability, as,

$$p_2(thresh) = \sum_{i \geq thresh} i \text{ intersections} = \sum_{i \geq thresh} \frac{\binom{tD+1}{i}\binom{N-(tD+1)}{tD+1-i}}{\binom{N}{i}}.$$

For our recursive estimates, we also estimated the probability that for a given set of size $tD+1$, and s other sets of size $tD+1$, the number of intersections between

the given set and the union of the s sets exceeds a threshold $thresh$. We did this by partitioning $thresh$ among the s sets, and calculating the probability that pairwise, there were at least that many intersections, using the above pairwise calculation. We then summed over all partitions of $thresh$.

$$s_q(thresh) \leq \sum_{\sum_{i=0}^{q-1} a_i = thresh} \prod_i p_2(a_i).$$

Now, for estimating p_q, for $q > 2$, we estimated this by picking one of our q sets, and,

$$p_q \leq \sum_{i=0}^{thresh} s_{q-1}(i) p_{q-1}(thresh - i).$$

Finally, to obtain our security parameters, we evaluated $p_q(t+1)$. In all these calculations, we used dynamic programming to make it somewhat faster, but did not concern ourselves with efficiency. These scripts are not practical to run for large q or N.

For a fixed q and D, for each value of N considered, we found the optimal value of t, and then found the probability that there would be more than t intersections. Table 5.9 gives values of N and t that give at least 20, 40, and 80 bits of security, for different values of q and D. Appendix A contains the data for N and t for the pairs of (q, D) studied.

Cover-Free Sets

For the GVW scheme to achieve simulation security, each key must have some ζ polynomial nonzero that none of the other keys have as nonzero. Each ciphertext has v nonzero ζ polynomials out of S total. We want to find the probability that this condition fails to hold.

We can rephrase this problem in terms of sets. Given q subsets s_i of size v of a set of size S, what is the probability that for each subset s_i, $s_i \not\subset \cup_{j \neq i} s_j$. These sets are referred to as cover-free sets [23].

		20 bits security		40 bits security		80 bits security	
q	D	N	t	N	t	N	t
2	2	210	14	430	29	850	59
2	3	540	14	1,110	30	2,220	60
2	4	1,000	14	2,100	30	4,300	62
2	5	1,700	15	3,400	30	7,000	63
2	6	2,400	14	5,000	30	10,400	64
3	2	750	13	1,650	29	3,400	62
3	3	1,700	12	3,800	29	8,000	62
3	4	3,200	12	6,800	28	14,400	62
4	2	1,700	13	3,600	29	7,400	63
4	3	3,900	13	8,100	29	16,800	62
5	2	3,000	12	6,200	28	12,800	63
6	2	4,500	11	9,900	28	19,800	62
7	2	6,400	11	14,000	27	28,800	62

Table 5.9: Values of N and t that achieve 20, 40, and 80 bits of security for small values of q and D.

We used a python script to estimate for given values of q, S, and v, the probability that the Δ, as defined in Section 3.3.2, for a functional key is covered by the Δs for $q-1$ other functional keys.

We started with a target set to be covered, of size a, one set to cover it, of size v, and threshold $thresh$ of the elements of the target that are covered, $p_1(thresh)$. (When there are two sets, distinguishing the target and covering sets strictly leads to overestimates, but it is more useful for more than two sets.) The probability of this occurring we calculated as,

$$p_1(thresh, a) = \sum_{i=thresh}^{a} \frac{\binom{a}{thresh}\binom{S-a}{v-thresh}}{\binom{S}{v}}.$$

Now, for more than one set covering the target, we estimated this probability recursively by partitioning the threshold.

$$p_q(thresh, a) \leq \sum_{i=0}^{thresh} p_1(i, a) p_{q-1}(thresh - i, a - i)$$

Then, we evaluated our estimate for each possible target, $q(p_{q-1}(v, v))$.

We then, for a given q and S, found the best value of v by finding the first local

	20 bits security		40 bits security		80 bits security	
q	S	v	S	v	S	v
2	24	12	45	22	86	43
3	63	16	117	30	228	59
4	100	18	185	34	360	67
5	134	18	253	36	480	69
6	180	20	320	37	620	73
7	220	20	390	38	740	73

Table 5.10: Values of S and v that achieve 20, 40, and 80 bits of security for small values of q and D.

minimum of the probability that the sets are not cover-free. Table 5.10 shows, for different values of q, the values of S and v that are required to give 20, 40, and 80 bits of security. Appendix B contains the data for S and v for the values of q studied.

5.2.2 Inner Product Circuits

We investigated the performance of the GVW scheme for different parameters and options. First, we used the parameters derived in Section 5.2.1 to evaluate the performance of the GVW scheme without Δ and the ζ polynomials at the 20, 40, and 80 bit security levels, for different number of keys supported, and different maximum depth of the circuits, for inner product circuits modulo 8123, with length one vectors, using AES128 as the base encryption scheme. The timing results are in Table 5.11, and the space results are in Table 5.12. We also give the timings and space usage for the SS one-key scheme as a comparison.

We do not give data for larger values of q and D for two reasons. First, using larger q or D, or higher security levels, requires increasing the prime from 8123 to a larger number. Second, increasing it also increases the amount of memory that is required, and the test computer could not handle larger numbers without significant performance degradation.

From these tables we see that using the GVW scheme instead of the SS scheme induces a large multiplicative slowdown. This means that the ciphertext for multiplying two numbers modulo 8123 is 21 Megabytes, to make the scheme 2-collusion secure, at the lowest security level. While that might be small enough to compute

q	D	Bits	N	t	Setup	KeyGen	Encrypt	Decrypt
2	2	20	210	14	37.625	0.690	89.560	4.777
2	2	40	430	29	73.778	1.378	181.696	9.737
2	2	80	850	59	156.066	2.999	393.102	20.627
2	3	20	540	14	93.664	1.593	234.796	9.720
2	3	40	1,110	30	188.416	3.454	473.119	19.941
2	3	80	2,220	60	382.954	7.222	1008.510	41.646
2	4	20	1,000	14	171.853	3.004	441.235	16.385
2	4	40	2,100	30	362.385	6.376	918.213	35.061
2	4	80	4,300	62	753.583	15.785	1987.400	74.010
2	5	20	1,700	15	287.266	5.156	733.898	25.442
2	5	40	3,400	30	569.548	11.380	1459.884	51.147
2	5	80	7,000	63	1168.522	24.119	2903.822	110.291
2	6	20	2,400	14	403.647	6.879	1020.871	33.973
2	6	40	5,000	30	835.307	16.765	2126.857	72.164
3	2	20	750	13	132.157	2.197	333.237	10.902
3	2	40	1,650	29	279.208	5.041	716.128	23.256
3	2	80	3,400	62	576.197	11.049	1465.432	48.541
3	3	20	1,700	12	285.230	4.816	721.994	21.824
3	3	40	3,800	29	638.692	12.321	1632.224	50.258
3	3	80	8,000	62	1331.735	26.141	3363.164	107.375
4	2	20	1,700	13	285.838	4.724	730.118	20.817
4	2	40	3,600	29	601.807	11.483	1544.990	45.138
4	2	80	7,400	63	1232.802	23.977	3103.926	95.758
4	3	20	3,900	13	649.961	12.102	1652.625	46.935
4	3	40	8,100	29	1344.793	25.802	3363.368	101.239
5	2	20	3,000	12	583.392	9.400	1302.697	36.615
5	2	40	6,200	28	1043.119	20.320	2607.043	76.374
6	2	20	4,500	11	756.816	13.778	1916.436	52.836
7	2	20	6,400	11	1066.648	19.921	2690.259	74.077
SS scheme					0.304	0.015	0.603	0.141

Table 5.11: GVW with SS with AES128 running times for different values of q and D with 20, 40, and 80 bits of security, with the base SS scheme as a reference. The column Bits is the number of bits of security offered. This was for an inner product mod 8123 circuit, with 1 element, or equivalently, multiplication mod 8123. This is for circuits not using Δ and ζ polynomials. All times are in milliseconds.

					MSK/MPK		FK	CT
q	D	Bits	N	t	size	keys	size	size
2	2	20	210	14	101,434	5,460	7,301	21,153,724
2	2	40	430	29	207,694	11,180	14,876	43,314,764
2	2	80	850	59	410,554	22,100	30,065	85,622,204
2	3	20	540	14	260,824	14,040	10,854	54,395,284
2	3	40	1,110	30	536,134	28,860	23,003	111,812,524
2	3	80	2,220	60	1,072,264	57,720	45,768	223,625,044
2	4	20	1,000	14	483,004	26,000	14,407	100,732,004
2	4	40	2,100	30	1,014,304	54,600	30,596	211,537,204
2	4	80	4,300	62	2,076,904	111,800	62,985	433,147,604
2	5	20	1,700	15	821,104	44,200	19,220	171,244,404
2	5	40	3,400	30	1,642,204	88,400	38,200	342,488,804
2	5	80	7,000	63	3,381,004	182,000	79,931	705,124,004
2	6	20	2,400	14	1,159,204	62,400	21,495	241,756,804
2	6	40	5,000	30	2,415,004	130,000	45,791	503,660,004
3	2	20	750	13	362,254	19,500	6,823	75,549,004
3	2	40	1,650	29	796,954	42,900	14,921	166,207,804
3	2	80	3,400	62	1,642,204	88,400	31,622	342,488,804
3	3	20	1,700	12	821,104	44,200	9,361	171,244,404
3	3	40	3,800	29	1,835,404	98,800	22,264	382,781,604
3	3	80	8,000	62	3,864,004	208,000	47,310	805,856,004
4	2	20	1,700	13	821,104	44,200	6,832	171,244,404
4	2	40	3,600	29	1,738,804	93,600	14,927	362,635,204
4	2	80	7,400	63	3,574,204	192,400	32,132	745,416,804
4	3	20	3,900	13	1,883,704	101,400	10,125	392,854,804
4	3	40	8,100	29	3,912,304	210,600	22,266	815,929,204
5	2	20	3,000	12	1,449,004	78,000	6,331	302,196,004
5	2	40	6,200	28	2,994,604	161,200	14,424	624,538,404
6	2	20	4,500	11	2,173,504	117,000	5,825	453,294,004
7	2	20	6,400	11	3,091,204	166,400	5,825	644,684,804
SS scheme					483	26	250	100,732

Table 5.12: GVW with SS with AES128 running times for different values of q and D with 20, 40, and 80 bits of security, with the base SS scheme as a reference. The column Bits is the number of bits of security offered. This was for an inner product mod 8123 circuit, with 1 element, or equivalently, multiplication mod 8123. This is for circuits not using Δ and ζ polynomials. All sizes are in bytes

	Length 1			Length 10		
Keys	20 bits	40 bits	80 bits	20 bits	40 bits	80 bits
2	2.130	3.085	4.873	1.117	1.210	1.408
3	3.904	6.361	11.222	1.295	1.598	2.193
4	5.587	9.454	18.248	1.503	1.979	2.923
5	7.134	12.547	25.079	1.693	2.360	3.587
6	9.226	16.229	33.049	1.951	2.736	4.361
7	11.046	20.268	39.881	2.175	3.128	5.025

Table 5.13: Ratios between ciphertext sizes for using Singleton FE and Δ and ζ polynomials for the given number of bits of security for inner product modulo 8123 circuits of short lengths, for the given number of keys for which the scheme is secure.

inner products for some short vectors, most of the values of q and D we studied were not able to give us much. This blowup, combined with the previously discussed large ciphertexts for the SS scheme, means that the GVW scheme is impractical in most cases.

5.2.3 Space Cost of Simulation Security

The GVW scheme ensures adaptive simulation security via using the SS scheme with the singleton FE as a base encryption scheme, and using Δ and the ζ polynomials. We now investigate how expensive this is for various parameters. Table 5.13 and Table 5.14 give the ratios between the sizes of the ciphertexts with 20, 40, or 80 bits of security as compared to ciphertexts which use neither the singleton scheme nor Δ and ζs, for inner product circuits modulo 8123 of lengths 1 and 10 for the first table, and 100 and 1000 for the second table, for different bounds on the number of keys for which it is secure.

Since using Δ and the ζ polynomials is just an additive factor, and as we saw in Section 5.1.1, the singleton scheme does not increase the ciphertext size that much for inner products, achieving adaptive simulation security is cheap for even moderately long inner products.

	Length 100			Length 1000		
Keys	20 bits	40 bits	80 bits	20 bits	40 bits	80 bits
2	1.019	1.028	1.038	1.001	1.002	1.004
3	1.036	1.060	1.101	1.003	1.005	1.010
4	1.053	1.091	1.160	1.004	1.008	1.016
5	1.068	1.121	1.213	1.006	1.011	1.021
6	1.089	1.152	1.275	1.008	1.014	1.027
7	1.107	1.183	1.328	1.010	1.017	1.032

Table 5.14: Ratios between ciphertext sizes for using Singleton FE and Δ and ζ polynomials for the given number of bits of security for inner product modulo 8123 circuits of long lengths, for the given number of keys for which the scheme is secure.

5.3 Stateful Functional Encryption

We now look at the performance of the stateful functional encryption scheme. As we discussed in Section 3.3.2, it is a scheme which can issue a bounded number of keys, but requires keeping state. We compare it to the GVW scheme to show the cost of avoiding having to keep state.

5.3.1 Inner Product Modulo a Prime

The Stateful FE scheme grows linearly with q, the limit on the number of keys it can to support. We compare the performance of equivalently sized stateful schemes, with 210, 430, and 850 keys, to the 20, 40, and 80 bit levels of security for the $q = 2$, $D = 2$ case. Table 5.15 shows the differences in running times, and Table 5.16 shows the space usage differences. We also include the stateful scheme with $q = 2$, to see the multiplicative increase in time and space usage between the schemes when only two keys are going to be given out.

For similar sized instances, the only difference in space usage is that the GVW scheme has larger functional keys due to the added complexity of the scheme. We also note that the stateful scheme is about as fast, except for Decryption, where it is twice as fast, while being able to support more keys. The Stateful scheme is significantly more practical than the GVW scheme for applications where keeping state is reasonable.

Scheme	Setup	KeyGen	Encrypt	Decrypt
Stateful, q=2	5.146	0.133	10.501	1.279
GVW, q=2, 20 bits	354.759	6.162	935.168	44.292
Stateful, q=210	352.841	4.780	981.477	21.626
GVW, q=2, 40 bits	727.505	12.651	2004.401	91.484
Stateful, q=430	724.993	9.604	1992.352	45.316
GVW, q=2, 80 bits	1435.318	26.464	3898.106	570.842
Stateful, q=850	1411.076	19.857	3644.260	159.140

Table 5.15: Running times for the GVW scheme and Stateful Scheme for similar space usage, for length 10 inner products modulo 8123 in milliseconds.

	MSK/MPK		FK	CT
Scheme	size	keys	size	size
Stateful, q=2	9,630	520	2,479	2,073,876
GVW, q=2, 20 bits	1,010,944	54,600	71,882	217,756,774
Stateful, q=210	1,010,944	54,600	2479	217,756,774
GVW, q=2, 40 bits	2,070,024	111,800	146,281	445,882,914
Stateful, q=430	2,070,024	111,800	2479	445,882,914
GVW, q=2, 80 bits	4,091,904	221,000	295,078	881,396,454
Stateful, q=850	4,091,904	221,000	2479	881,396,454

Table 5.16: Space usage for the GVW scheme and Stateful Scheme for similar space usage in bytes.

	Encrypt		Decrypt		Ciphertext Size	
Bits	SS	CFE	SS	CFE	SS	CFE
10,000	0.168	(3.81,0.30)	0.030	(2.19,0.31)	2520.8KB	919.0KB
16,000	0.265	(3.95,0.47)	0.048	(2.70,0.45)	4245.9KB	1470.6KB
20,000	0.342	(6.19,0.55)	0.060	(4.70,0.52)	5405.3KB	1838.4KB
60,000	1.005	(11.15,1.52)	0.178	(88.34,0.44)	16787.3KB	5515.9KB
1,500,000	25.109	(469.78,45.81)	4.476	(426.50,44.26)	422.0MB	135.0MB

Table 5.17: Comparison between our SS scheme implementation and Controlled Functional Encryption, for Hamming distance. Times for Controlled Functional Encryption are (offline, online). All times are in seconds.

5.4 Comparisons to Controlled Functional Encryption

We also compare our work to controlled functional encryption [54]. This allows us to both make an apples to apples comparison to ensure that our work is comparable in some cases, and measure the performance trade-off of the extra interaction used by controlled functional encryption.

We compare the SS scheme against Controlled Functional Encryption for Hamming distance. As we can see from Table 5.17, our implementation is about twice as fast as their online phase for encryption, and about ten times as fast for decryption. However, our encryption algorithm also generates the garbled circuit, which both their encrypt and decrypt perform offline. Part of the reason for the speedup is probably due to using libgarble, which both takes advantage of more features, and is written in C++ as opposed to Java [36].

Our implementation has ciphertexts about three times as large. Part of this is caused by the fact that we include the circuit description for the circuit being garbled in the ciphertext, instead of regenerating it in the Decrypt algorithm, or doing it offline and storing it. Additionally, as the SS scheme is for functional encryption, our implementation also includes encrypted labels for the circuit description.

Also, we note that both the stateful and GVW schemes use a one-key functional encryption scheme as a black box, they each run multiplicatively slower, and take multiplicatively more space, compared to the SS scheme. Given this, we do not

explicitly compare them.

Chapter 6

Future Work

While FIFE is a good start at investigating functional encryption, there are many ways in which it can be improved. We summarize some of these directions below.

6.1 Additional Functional Encryption Schemes

One direction in which to continue this work would be to add more functional encryption schemes. This would lead to both understanding the practical implications of more functional encryption schemes, and understand the performance trade-offs between the different schemes.

6.1.1 GKPVZ Scheme

One such scheme is the GKPVZ scheme, discussed in Section 3.3.1. This is another one-key functional encryption scheme. It uses techniques from lattice cryptography to output ciphertexts that do not grow with the size of the circuit being computed. However, it also relies on fully homomorphic encryption and attribute-based encryption to do so.

There have been a few implementations of fully homomorphic encryption. Gentry and Halevi implemented it in 2010, with large keys, and bootstrapping operations that took minutes [28]. Halevi and Shoup made HElib in 2014 as a more efficient

implementation of fully homomorphic encryption [37]. However, these implementations of fully homomorphic encryption are still far from practical. In this case, there is a clear trade-off between asymptotic performance and practicality, and it deserves further study.

6.1.2 Functional Encryption Based on Function-Hiding Inner Product Encryption

We also want to compare the performance of the one-key scheme proposed by Kim et al. [44]. This is another one-key functional encryption scheme, but this one is based on function-hiding inner product encryption, so it would be interesting to see what are the advantages and disadvantages of the different type of construction. Also, as their scheme is a two input functional encryption scheme, it will be interesting to see what the cost is of this variant of functional encryption.

6.1.3 Agrawal-Rosen Scheme

Another scheme to consider is the Agrawal-Rosen scheme, which we discussed in 3.3.2. This is a bounded-collusion scheme, which additionally is able to improve the speed of the Encrypt algorithm by moving much of the computation to an offline setup. This scheme also achieves better asymptotic growth compared to the GVW scheme as the number of functional keys under which it is secure increases. It would be interesting to investigate both how the size of its keys and ciphertexts, and the running times of its algorithms, grow in practice, but also how much of an effect moving computation offline is, and the practical gains realizable by that difference.

6.2 Other Traditional Encryption Schemes

While we investigated RSA and AES, there are other choices of public-key and secret-key schemes that can be used. Especially since we determined in Section 5.1.1 that RSA was significantly slower than AES, it would be especially useful to investigate

other, potentially faster, public-key schemes, such as Elliptic Curve Cryptography [46, 52], or ElGamal [22] which each have their own advantages. Another possibility is to use a lattice-based public-key encryption scheme such as Regev's cryptosystem [57]. At the expense of efficiency, this has the advantage that there is no known quantum attack on it, unlike the previous schemes [68].

6.3 Optimizations

6.3.1 Garbled Arithmetic Circuits

The GVW scheme discussed in Section 3.3.2 encodes polynomials over finite fields. When combined with the SS one-key scheme, this means doing field operations with a binary circuit. While this is possible, it requires large circuits to compute, as discussed in Section 5.1.5. It would be more convenient if we could operate directly on the field elements. The garbled arithmetic circuits of Applebaum, Ishai, and Kushilevitz could allow us to do this [5]. However, we were not able to figure out how to use their scheme without leaking information.

6.3.2 Supporting More Circuits

We only included implementations for a few circuits, as shown in table 4.1. Supporting more circuits would give a better idea of what functions are practical in the existing schemes.

A more general approach would be to use general universal circuits, as we discussed in Section 3.2.4. This would allow us to create functional encryption schemes that support all functions.

6.3.3 Reducing Circuit Size

As we discussed in Section 5.1.5, a large part of the ciphertext's size came from the size of the circuit. Reducing the size of the circuits would improve the performance of the scheme.

One specific way to improve the performance would be to use a more efficient multiplication circuit. Currently, as discussed in Section 4.2, we are using a multiplication circuit that uses a quadratic number of gates in the input lengths, and more efficient multiplication algorithms exist [41, 63].

6.3.4 Implicit Circuit Descriptions

As currently implemented, each ciphertext currently includes a description of the universal circuit that is garbled. The ciphertext size could be somewhat reduced by not storing this information, and instead just having it publicly available. Additionally, for the GVW scheme, even if not totally removed from the ciphertext, we could store it just once for the entire GVW scheme ciphertext, instead of once per instance of the SS ciphertext it contains.

6.3.5 Parallelism

Currently, FIFE runs in a single-threaded manner. Using parallelism could speed up some algorithms and schemes significantly. When running the setup for the SS scheme, we generate a lot of key pairs for the base encryption scheme, which could easily be done in parallel.

For the GVW scheme, all of the algorithms can benefit from parallelism. The Setup algorithm runs the Setup algorithm for many copies of a one-key functional encryption scheme. The KeyGen algorithm again runs the one-key functional encryption KeyGen algorithm many times. The Encrypt algorithm runs many copies of the one-key Encrypt algorithm. The Decrypt algorithm runs many copies of the one-key Decrypt algorithm, and then just does polynomial interpolation. All of these likely can achieve higher speeds via parallelism.

6.3.6 Streaming Garbled Circuits

As discussed in Section 5.1.5, constructing the garbled circuits we are using often takes significantly more memory than when written to file. Also, as we saw in Sections

5.1 and 5.2, the ciphertexts written to file can already be quite large. As libgarble requires holding the circuit in memory to do operations, this scheme cannot support significantly larger circuits or higher values of q, the limit on collusion. One way to get around this is to use a streaming garbled circuit, which only needs to load part of the circuit into memory at a given time. Adding this feature to libgarble would increase the bound on what could be supported.

Chapter 7

Conclusion

We built a Framework for Investigating Functional Encryption, and investigated the performance of various functional encryption schemes. We implemented some of the existing functional encryption schemes, and evaluated their performance in detail, for a few classes of interesting circuits, and showed the practical limits for each of them.

We found that for the Sahai-Seyalioglu scheme, secret-key functional encryption is significantly faster than functional encryption with public keys, especially in the Setup phase. However, for even moderately complex circuit families, the extra space used by public-key encryption is not significant. If the Setup phase is done in an offline fashion, and the needed public-key encryption key pairs are just loaded, then using public-key encryption is a reasonable approach.

We found that the biggest constraining factor for the GVW scheme using the SS scheme, and to a lesser extent for the SS scheme itself, was the size of the ciphertexts. In the GVW scheme, in order to get sufficient security for relatively small values of q, the base one-key scheme needs to be repeated hundreds or thousands of times. This means that it is not practical even for just multiplying two number modulo 8123 if you want to be secure against a 7-key collusion. For the SS scheme, we were able to do better, and take the inner product of vectors of length 10000 modulo 8123, or inner product modulo a prime with 31 bits for a shorter vector, but the ciphertexts for these also grow large. In comparison, the running times were often more manageable.

While our work does not offer the most compelling running times and key and

circuit sizes for functional encryption, it provides a first implementation of functional encryption schemes, and highlights some of the performance bottlenecks in various implementations. Additionally, for the GVW scheme in particular, it provides a limited analysis of the concrete relationship between parameters for the scheme and security, for some small instances of the scheme.

Our work also creates an extendible framework to examine the performance of additional functional encryption schemes and compare them to other schemes. It also allows different components to be added, and their impact on the functional encryption schemes to be easily evaluated.

FIFE is a first step in the work to make functional encryption usable in practice. There are many places where functional encryption could offer a better security relationship for data. Just as public-key encryption is now heavily utilized and integral to the tech industry, functional encryption could create or enable new ways of sharing, storing, and using data for many people. However, more work is needed to keep moving in that direction.

Bibliography

[1] Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

[2] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 500–518. Springer, Heidelberg, August 2013.

[3] Shweta Agrawal and Alon Rosen. Online-offline functional encryption for bounded collusions. Cryptology ePrint Archive, Report 2016/361, 2016. `http://eprint.iacr.org/`.

[4] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Computationally private randomizing polynomials and their applications. *Computational Complexity*, 15(2):115–162, 2006.

[5] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. In Rafail Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 120–129. IEEE Computer Society Press, October 2011.

[6] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. Cryptology ePrint Archive, Report 2011/136, 2011. `http://eprint.iacr.org/2011/136`.

[7] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[8] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, Heidelberg, May 1995.

[9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM Press, May 1988.

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

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

[12] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, Heidelberg, August 2001.

[13] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, Heidelberg, March 2011.

[14] Xavier Boyen and Brent Waters. Anonymous hierarchical identity-based encryption (without random oracles). In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 290–307. Springer, Heidelberg, August 2006.

[15] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, Heidelberg, August 2011.

[16] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *28th Annual ACM Symposium on Theory of Computing*, pages 639–648. ACM Press, May 1996.

[17] Clifford Cocks. An identity based encryption scheme based on quadratic residues. In Bahram Honary, editor, *8th IMA International Conference on Cryptography and Coding*, volume 2260 of *Lecture Notes in Computer Science*, pages 360–363. Springer, Heidelberg, December 2001.

[18] Wei Dai. Crypto++ library 5.6.3. `https://www.cryptopp.com/`, 2016. Accessed May 15, 2016.

[19] Pratish Datta, Ratna Dutta, and Sourav Mukhopadhyay. Functional encryption for inner product with full function privacy. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 9614 of *Lecture Notes in Computer Science*, pages 164–195. Springer, Heidelberg, March 2016.

[20] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 519–535. Springer, Heidelberg, August 2013.

[21] Billy Donahue. Google test. `https://github.com/google/googletest`, 2016. Accessed May 15, 2016.

[22] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, Heidelberg, August 1984.

[23] Paul Erdös, Peter Frankl, and Zoltán Füredi. Families of finite sets in which no set is covered by the union ofr others. *Israel Journal of Mathematics*, 51(1):79–89, 1985.

[24] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, March 2004.

[25] Sadayuki Furuhashi. Messagepack. `http://msgpack.org/index.html`, 2016. Accessed May 15, 2016.

[26] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Functional encryption without obfuscation. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A: 13th Theory of Cryptography Conference, Part II*, volume 9563 of *Lecture Notes in Computer Science*, pages 480–511. Springer, Heidelberg, January 2016.

[27] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[28] Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, Heidelberg, May 2011.

[29] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 197–206. ACM Press, May 2008.

[30] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 555–564. ACM Press, June 2013.

[31] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 365–377. ACM, 1982.

[32] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 162–179. Springer, Heidelberg, August 2012.

[33] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Attribute-based encryption for circuits. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 545–554. ACM Press, June 2013.

[34] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Predicate encryption for circuits from LWE. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 503–523. Springer, Heidelberg, August 2015.

[35] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 06: 13th Conference on Computer and Communications Security*, pages 89–98. ACM Press, October / November 2006. Available as Cryptology ePrint Archive Report 2006/309.

[36] Adam Groce, Alex Ledger, Alex J. Malozemoff, and Arkady Yerukhimovich. Compgc: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive, Report 2016/458, 2016. `http://eprint.iacr.org/`.

[37] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, Heidelberg, April 2015.

[38] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 451–462. ACM Press, October 2010.

[39] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[40] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st Annual Symposium on Foundations of Computer Science*, pages 294–304. IEEE Computer Society Press, November 2000.

[41] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[42] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[43] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 146–162. Springer, Heidelberg, April 2008.

[44] Sam Kim, Kevin Lewi, Avradip Mandal, Hart Montgomery, Arnab Roy, and David J. Wu. Function-hiding inner product encryption is practical. Cryptology ePrint Archive, Report 2016/440, 2016. `http://eprint.iacr.org/`.

[45] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. Cryptology ePrint Archive, Report 2016/093, 2016. `http://eprint.iacr.org/2016/093`.

[46] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[47] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS 09: 8th International Conference on Cryptology and Network Security*, volume 5888 of *Lecture Notes in Computer Science*, pages 1–20. Springer, Heidelberg, December 2009.

[48] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, Heidelberg, July 2008.

[49] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In Gene Tsudik, editor, *FC 2008: 12th International Conference on Financial Cryptography and Data Security*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, Heidelberg, January 2008.

[50] 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 Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 62–91. Springer, Heidelberg, May 2010.

[51] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[52] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPTO'85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, Heidelberg, August 1986.

[53] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 129–139, New York, NY, USA, 1999. ACM.

[54] Muhammad Naveed, Shashank Agrawal, Manoj Prabhakaran, XiaoFeng Wang, Erman Ayday, Jean-Pierre Hubaux, and Carl A. Gunter. Controlled functional encryption. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 1280–1291. ACM Press, November 2014.

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

[56] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, Heidelberg, December 2009.

[57] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM Press, May 2005.

[58] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *FOUNDATIONS OF SECURE COMPUTATIONS*, page 171, 1979.

[59] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978.

[60] Kurt Rohloff. Palisade. `https://git.njit.edu/groups/palisade`, 2016. Accessed May 15, 2016.

[61] Amit Sahai and Hakan Seyalioglu. Worry-free encryption: functional encryption with public keys. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 463–472. ACM Press, October 2010.

[62] Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer, Heidelberg, May 2005.

[63] Doz Dr A Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[64] Gadiel Seroussi. *Table of low-weight binary irreducible polynomials*. 1998.

[65] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[66] Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer, Heidelberg, August 1984.

[67] Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 457–473. Springer, Heidelberg, March 2009.

[68] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[69] Victor Shoup. NTL: A library for doing number theory. `https://www.cryptopp.com/`, 2016. Accessed May 15, 2016.

[70] Gaurav Singh. FIFE: A Framework for Investigating Functional Encryption. `https://github.com/gauravjsingh/FIFE`, 2016. Accessed May 17, 2016.

[71] Leslie G Valiant. Universal circuits (preliminary report). In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 196–203. ACM, 1976.

[72] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 678–697. Springer, Heidelberg, August 2015.

[73] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, October 1986.

[74] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, Heidelberg, April 2015.

Appendix A

Small-Intersection Sets Data

In Section 5.2.1, we discussed how we estimated the bits of security given for values of q, D, N, and t. The data we generated is below.

N	t	Bits of Security
10	1	0.157100961085
30	1	2.44746449903
50	3	4.53449517872
70	4	6.53804152127
90	6	8.5474765021
110	7	10.533037078
130	9	12.4964461256
150	10	14.4696247209
170	11	16.4166854921
190	13	18.3700843058
210	14	20.314047658
230	16	22.2462602285
250	17	24.1878758877
270	18	26.1152685884
290	20	28.0448449247
310	21	29.9721458588
330	23	31.8891417167
350	24	33.8163609647
370	25	35.7339445069
390	27	37.6507129557
410	28	39.56904706
430	29	41.4795956509
450	31	43.3961326831
470	32	45.3078523632
490	34	47.2166551851
510	35	49.129388495
530	36	51.0359908338
570	39	54.8531464031
610	42	58.6654260709
650	45	62.4735299409
690	47	66.2791522412
730	50	70.0859650426
770	53	73.8894038584
810	56	77.6898712326
850	59	81.4877027294
890	61	85.287005466

Table A.1: Bits of security for given values of N and optimal values of t when $q = 2$, $D = 2$.

N	t	Bits of Security
150	4	6.01165833582
180	4	7.15412830618
210	5	8.30749793806
240	6	9.44245835527
270	7	10.5637950687
300	8	11.6746390236
330	9	12.7771509386
360	9	13.8773248275
420	11	16.0832178391
480	13	18.264595022
540	14	20.4349727801
600	16	22.6070573889
660	18	24.7645723918
720	19	26.9188055
780	21	29.0724808374
840	23	31.2161262486
900	24	33.3606639928
960	26	35.5027582634
1020	28	37.6373344144
1080	29	39.7754023077
1110	30	40.8432720155
1140	31	41.9095385051
1170	32	42.9743194338
1200	33	44.0377205638
1260	34	46.1711631952
1320	36	48.2994939391
1440	39	52.5528964928
1560	42	56.7994170959
1680	46	61.0442646103
1800	49	65.2862154631
1860	51	67.403831879
1920	52	69.5232663359
1980	54	71.6416352399
2040	56	73.7569338608
2100	57	75.8751865212
2160	59	77.9912876822
2220	60	80.1050296239
2280	62	82.2218992842

Table A.2: Bits of security for given values of N and optimal values of t when $q = 2$, $D = 3$.

N	t	Bits of Security
400	5	8.61902890009
500	7	10.5834665939
600	8	12.5083560961
700	10	14.4257535703
800	11	16.3253918744
900	13	18.2166325069
1000	14	20.1011792659
1100	16	21.9757494549
1200	17	23.85024976
1300	19	25.7132622768
1400	20	27.5805903519
1500	22	29.4351166597
1600	23	31.29706631
1700	24	33.1471375984
1800	26	35.0028625287
1900	27	36.8502124872
2000	29	38.7001782855
2100	30	40.5453262442
2200	32	42.3905962689
2400	35	46.0752935505
2600	38	49.7551693757
2800	41	53.4309262737
3000	43	57.1037841565
3200	46	60.7744197057
3400	49	64.4421364579
3600	52	68.1072736782
3800	55	71.7701146916
3900	57	73.5997546318
4000	58	75.4308985471
4100	60	77.259016811
4200	61	79.0898287908
4300	62	80.9167406441
4400	64	82.7470801685

Table A.3: Bits of security for given values of N and optimal values of t when $q = 2$, $D = 4$.

N	t	Bits of Security
200	1	3.11679390793
400	3	5.73514591332
600	5	8.20038492117
800	7	10.5962211332
1000	9	12.952191145
1200	10	15.2872713692
1400	12	17.6118084529
1600	14	19.9195030578
1700	15	21.0684166734
1800	16	22.2145746357
2000	18	24.499828614
2200	20	26.777226024
2400	21	29.0546272491
2600	23	31.3271179179
2800	25	33.5940309046
3000	27	35.8562301107
3200	29	38.1143934984
3400	30	40.3708759784
3600	32	42.6277807516
3800	34	44.8814003511
4000	36	47.1321299867
4200	38	49.38029816
4400	40	51.6261807847
4600	41	53.8735972644
4800	43	56.1194700794
5000	45	58.3633314301
5200	47	60.6053714672
5400	49	62.8457549613
5600	50	65.0859494058
5800	52	67.3267485786
6000	54	69.5660354907
6200	56	71.8039329297
6400	58	74.0405496057
6600	60	76.2759821934
6800	61	78.5129480171
7000	63	80.7489692567
7200	65	82.9838788216

Table A.4: Bits of security for given values of N and optimal values of t when $q = 2$, $D = 5$.

N	t	Bits of Security
400	2	4.20240492792
800	4	7.58478717517
1200	7	10.8493674187
1600	9	14.0168709016
2000	12	17.1603535722
2400	14	20.2658020801
2800	17	23.360816801
3200	19	26.4364635901
3600	22	29.505269257
4000	24	32.5633726556
4400	27	35.6157389315
4800	29	38.662275633
5200	32	41.7033624124
5600	34	44.7416967055
6000	37	47.7745621833
6400	39	50.8067758506
6800	42	53.83338216
7200	44	56.8608532206
7600	47	59.8825345721
8000	49	62.9062225215
8400	52	65.9239274211
8800	54	68.9445272011
9200	56	71.9597196734
9600	59	74.9769852108
10000	61	77.9901427119
10400	64	81.0045243181
10500	64	81.7559709448

Table A.5: Bits of security for given values of N and optimal values of t when $q = 2$, $D = 6$.

N	t	Bits of Security
200	2	7.5153259993
300	4	9.91739897296
400	6	12.2781967561
500	8	14.6193471168
600	10	16.947021665
650	11	18.1067840282
700	12	19.2642006412
750	13	20.419498693
800	13	21.5775248772
900	15	23.8880805298
1000	17	26.1907275923
1100	19	28.4870063365
1200	21	30.7780396666
1300	23	33.0646727582
1400	25	35.3475580572
1450	25	36.4892031771
1500	26	37.6312774476
1550	27	38.772455828
1600	28	39.9128024503
1650	29	41.0523746537
1700	30	42.1912239183
1750	31	43.3293966437
1800	32	44.466934803
1900	34	46.740256409
2000	36	49.01145498
2200	39	53.5516785401
2400	43	58.0884764872
2600	47	62.6190720309
2800	50	67.1446924684
3000	54	71.6708500376
3200	58	76.1925722705
3400	62	80.7104891166
3500	63	82.9685508755

Table A.6: Bits of security for given values of N and optimal values of t when $q = 3$, $D = 2$.

N	t	Bits of Security
500	3	8.40604528504
600	3	9.40939899867
700	4	10.3994780355
800	5	11.3797752594
900	6	12.3542778286
1000	6	13.334718623
1100	7	14.3273797914
1200	8	15.3122364582
1300	9	16.2912622865
1400	10	17.2657379557
1500	10	18.2457191105
1600	11	19.2294482591
1700	12	20.2081493266
1800	13	21.182685985
2000	14	23.129928963
2200	16	25.0809030946
2400	18	27.0192207305
2600	19	28.9646160146
2800	21	30.9018859982
3000	22	32.8369875157
3200	24	34.7732147498
3400	26	36.7013550683
3600	27	38.6356023732
3700	28	39.6003803185
3800	29	40.563467189
4000	30	42.490764008
4400	34	46.3401490561
4800	37	50.1892310241
5200	40	54.0333145109
5600	43	57.8731466848
6000	46	61.709318125
6400	49	65.5423038462
6800	53	69.3754829837
7200	56	73.2062735483
7600	59	77.0345366185
7800	61	78.9470966329
8000	62	80.8605462131
8200	64	82.7735187576
8400	65	84.6845341512

Table A.7: Bits of security for given values of N and optimal values of t when $q = 3$, $D = 3$.

N	t	Bits of Security
800	2	8.18643603908
1200	4	10.2700068194
1600	5	12.4037170797
2000	7	14.5628579368
2400	9	16.7122704729
2800	11	18.8545368871
3000	12	19.9233744957
3200	12	20.9936332165
3400	13	22.067673162
3600	14	23.1394462448
4000	16	25.2772495399
4400	18	27.408719052
4800	20	29.5350344481
5200	21	31.6662894776
5600	23	33.7930877959
6000	25	35.9154575061
6400	27	38.0340894131
6600	27	39.0932539279
6800	28	40.1543016198
7200	30	42.2736430709
7600	32	44.3897380623
8000	34	46.5030096425
8400	35	48.6157746459
8800	37	50.7299173048
9200	39	52.8415470052
9600	41	54.9509437967
10000	42	57.0583756604
10400	44	59.1687142351
10800	46	61.2770250377
11200	48	63.3835055016
11600	50	65.4883260161
12000	51	67.5943773261
12400	53	69.700130084
12800	55	71.8043408444
13200	57	73.9071379099
13600	58	76.0098238291
14000	60	78.1135467912
14400	62	80.2159427095
14800	64	82.3171114322

Table A.8: Bits of security for given values of N and optimal values of t when $q = 3$, $D = 4$.

N	t	Bits of Security
500	3	8.21232353446
600	3	9.23701258522
700	4	10.3081418689
800	5	11.3600740912
900	6	12.4031144514
1000	7	13.4424939867
1100	8	14.4809007183
1200	8	15.5362844901
1300	9	16.5903396598
1400	10	17.6423215351
1500	11	18.6932383072
1600	12	19.7436773259
1700	13	20.7939712248
1800	14	21.8442976088
2000	15	23.9608434367
2200	17	26.0742015413
2400	19	28.185229633
2600	21	30.2944457787
2800	22	32.411649416
3000	24	34.5262719211
3200	26	36.6382475884
3400	28	38.7479147445
3500	28	39.8039369428
3600	29	40.8611893392
3700	30	41.9177398668
3800	31	42.9736337356
4000	33	45.0836102422
4400	36	49.3003568171
4800	40	53.5174623183
5200	43	57.7282228803
5600	47	61.9404643729
6000	51	66.1461265599
6400	54	70.3542274425
6800	58	74.5566872084
7200	61	78.7603069222
7400	63	80.860765038
7600	65	82.9600816124
8000	68	87.1599883708

Table A.9: Bits of security for given values of N and optimal values of t when $q = 4$, $D = 2$.

N	t	Bits of Security
600	1	6.41685418585
1200	3	9.02389722148
1800	5	11.7009548643
2400	7	14.3959346874
3000	9	17.1036912004
3600	11	19.8206943237
3900	13	21.1834004661
4200	14	22.5551130754
4800	16	25.2981423084
5400	18	28.0402563263
6000	21	30.7836046459
6600	23	33.5353669904
7200	25	36.2833632925
7800	27	39.0279852534
8100	29	40.4040127474
8400	30	41.7791721711
9000	32	44.5265154355
9600	34	47.2702899199
10200	37	50.0173615204
10800	39	52.7623219984
11400	41	55.5040238625
12000	44	58.2472388036
12600	46	60.9896157035
13200	48	63.7291246887
13800	51	66.4690633859
14400	53	69.2090848095
15000	55	71.9465938382
15600	58	74.6838209371
16200	60	77.4217950512
16800	62	80.157555347
17400	65	82.8925381463

Table A.10: Bits of security for given values of N and optimal values of t when $q = 4$, $D = 3$.

N	t	Bits of Security
400	1	5.40021393707
800	2	7.81847117837
1200	4	10.2139854764
1600	6	12.5610510099
2000	8	14.9043999343
2400	9	17.2722131525
2800	11	19.6457165575
3000	12	20.8351560604
3200	13	22.0267993357
3600	15	24.4169934206
4000	17	26.8161271625
4400	19	29.2233675591
4800	21	31.6376754755
5200	23	34.0580065542
5600	25	36.4833961295
6000	27	38.9129898007
6200	28	40.129127851
6400	29	41.3460485363
6800	31	43.7819422676
7200	34	46.2207806106
7600	36	48.6627351203
8000	38	51.1060032163
8400	40	53.5502903319
8800	42	55.9953541947
9200	44	58.4409959324
9600	46	60.8870526563
10000	48	63.3333912904
10400	50	65.7799034423
10800	52	68.2265011454
11200	54	70.6731133223
11600	56	73.1196828518
12000	58	75.5661641333
12400	61	78.0131439012
12800	63	80.4605030475
13200	65	82.9076151378

Table A.11: Bits of security for given values of N and optimal values of t when $q = 5$, $D = 2$.

N	t	Bits of Security
600	1	5.36777630403
1200	2	7.75200002816
1800	4	10.0617401033
2400	5	12.3217856783
3000	7	14.6059797259
3600	9	16.8681434856
3900	10	17.9979878169
4200	10	19.129001605
4500	11	20.2729365559
4800	12	21.41650563
5400	14	23.7060696082
6000	16	26.0022240816
6600	18	28.3069207429
7200	20	30.6207809884
7800	22	32.9437302106
8400	24	35.2753358964
9000	26	37.6149936052
9300	27	38.7876327091
9600	28	39.9620309995
9900	29	41.1381033426
10200	30	42.3157661212
10800	32	44.6755394505
11400	34	47.0407306841
12000	36	49.4107665341
12600	38	51.7851232578
13200	40	54.1633261419
13800	42	56.5449472849
14400	44	58.9296024969
15000	46	61.3169478129
15600	48	63.7066759171
16200	50	66.0985126541
16800	52	68.4922137242
17400	54	70.8875616122
18000	56	73.2843627705
18600	58	75.6824450586
19200	60	78.0816554301
19800	62	80.4818578531
20400	64	82.8829314466

Table A.12: Bits of security for given values of N and optimal values of t when $q = 6$, $D = 2$.

N	t	Bits of Security
800	1	5.20896592233
1600	2	7.50385403636
2400	4	9.62480453888
3200	5	11.791520608
4000	6	13.8853324357
4800	8	16.0077908715
5600	9	18.096381049
6000	10	19.1596608004
6400	11	20.2170539857
6800	12	21.2707923171
7200	13	22.3225041738
8000	14	24.4398103303
8800	16	26.5655498741
9600	18	28.6909862209
10400	19	30.8286471872
11200	21	32.9758854642
12000	23	35.1267309467
12800	25	37.2826286944
13600	26	39.4490958952
14000	27	40.5358149646
14400	28	41.6237031114
15200	30	43.803200207
16000	32	45.9878564483
16800	34	48.1777477365
17600	36	50.3728208125
18400	37	52.5751393369
19200	39	54.7829634873
20000	41	56.9949118426
20800	43	59.2108344651
21600	45	61.4305548358
22400	47	63.6538811787
24000	51	68.110554352
25600	55	72.5792666499
26400	57	74.8176604971
27200	58	77.0594474999
28000	60	79.3035263707
28800	62	81.5496297584
29600	64	83.7976212592

Table A.13: Bits of security for given values of N and optimal values of t when $q = 7$, $D = 2$.

Appendix B

Cover-Free Sets Data

In Section 5.2.1, we discussed how we estimated the bits of security given for values of q, S, and v. The data we generated is below.

S	v	Bits of Security
10	5	6.9772799235
12	6	8.85174904142
14	7	10.7448338375
16	8	12.6517244331
18	9	14.5692622729
20	10	16.4952616915
22	11	18.4281474956
24	12	20.3667469509
26	13	22.3101634226
28	14	24.2576960027
30	15	26.2087864022
32	16	28.1629827126
34	17	30.1199139907
36	18	32.0792720062
38	19	34.0407978584
40	20	36.0042719824
42	21	37.9695065642
44	22	39.9363397003
46	23	41.9046308405
48	24	43.8742571915
50	25	45.8451108458
52	26	47.8170964697
54	27	49.7901294221
56	28	51.7641342135
58	29	53.7390432326
60	30	55.7147956863
62	31	57.6913367135
64	32	59.668616637
66	33	61.6465903307
68	34	63.6252166799
70	35	65.6044581197
72	36	67.5842802378
74	37	69.564651431
76	38	71.5455426081
78	39	73.5269269299
80	40	75.5087795832
82	41	77.4910775815
84	42	79.47379959
86	43	81.4569257715
88	44	83.4404376487

Table B.1: Bits of security for S and optimal values of v, for $q = 2$.

S	v	Bits of Security
30	8	8.81267574049
36	9	11.0027863492
42	11	13.194647269
48	12	15.3772726547
54	14	17.5747476782
57	15	18.6664593676
60	16	19.754540022
63	16	20.8557591955
66	17	21.9539943639
72	19	24.1395490169
78	20	26.3327803307
84	22	28.5225089661
90	23	30.7112894592
96	25	32.9041797852
102	26	35.0896192688
108	28	37.2849869549
111	29	38.3791883051
114	30	39.4713421692
117	30	40.5676449915
120	31	41.6651872684
120	31	41.6651872684
126	33	43.8541729993
132	34	46.0449448923
138	36	48.2361307811
144	37	50.4243696764
150	39	52.6174225852
156	41	54.8041042011
162	42	56.9981947921
168	44	59.1868015855
174	45	61.3785538101
180	47	63.5688479167
186	48	65.7585788479
192	50	67.9503636826
198	51	70.1383300835
204	53	72.3314413658
210	55	74.5198182511
216	56	76.7121531338
222	58	78.9018984588
228	59	81.0925561271
234	61	83.2835402957

Table B.2: Bits of security for S and optimal values of v, for $q = 3$.

S	v	Bits of Security
40	7	6.69019521984
50	9	9.02855097372
60	11	11.3750967355
70	12	13.727985077
80	14	16.0906335508
90	16	18.4521182656
95	17	19.6325179025
100	18	20.8127290679
105	19	21.9927768278
110	20	23.1726828492
120	22	25.5321417161
130	24	27.8912254976
140	26	30.250022245
150	27	32.6120117619
160	29	34.9749039138
170	31	37.3370772069
175	32	38.5179368301
180	33	39.6986636834
185	34	40.8792696496
190	35	42.0597652486
200	37	44.4204617052
210	39	46.780816382
220	40	49.1410568241
230	42	51.504094821
240	44	53.8666257428
250	46	56.2287152998
260	48	58.5904183553
270	50	60.951781069
280	52	63.3128425527
290	54	65.673636163
300	55	68.0351533899
310	57	70.3978319974
320	59	72.7601602537
330	61	75.1221721228
340	63	77.4838973248
350	65	79.8453619778
360	67	82.2065891262
370	69	84.5675991797

Table B.3: Bits of security for S and optimal values of v, for $q = 4$.

S	v	Bits of Security
50	6	5.46937983582
64	8	7.85271157004
78	10	10.2621395532
92	12	12.6878311125
106	14	15.1235651325
120	16	17.5655424473
127	17	18.7881102811
134	18	20.0114530319
141	19	21.2354181136
155	22	23.6848964451
169	24	26.1375397755
183	26	28.5907917115
197	28	31.0444507323
211	30	33.4983905376
225	32	35.9525305208
239	34	38.4068181179
246	35	39.6340058052
253	36	40.8612181864
260	37	42.0884525388
267	38	43.3157065692
281	40	45.77026616
300	43	49.1007179951
310	44	50.85452541
320	46	52.6070991916
330	47	54.3616306821
340	49	56.1136076156
350	50	57.8687694343
360	52	59.6202231057
370	53	61.375938495
380	54	63.1278427275
390	56	64.8831350989
400	57	66.6357081024
410	59	68.3903567622
420	60	70.1435296364
430	62	71.8976012336
440	63	73.6513151206
450	65	75.4048664716
460	66	77.1590707029
470	68	78.9121506286
480	69	80.6668012813
490	71	82.4194520374

Table B.4: Bits of security for S and optimal values of v, for $q = 5$.

S	v	Bits of Security
60	6	4.65846359932
80	8	7.30752195491
100	10	9.99568401724
120	13	12.7226319224
140	15	15.4744228095
160	17	18.2310190476
170	19	19.619729029
180	20	21.008835037
190	21	22.3974337711
200	22	23.7855575144
220	25	26.568133489
240	27	29.3547457171
260	30	32.138711347
280	32	34.9306645439
300	34	37.7187760619
320	37	40.5098698003
340	39	43.3011847658
360	42	46.0908370358
380	44	48.8844132815
400	46	51.6748434416
420	49	54.4681265805
440	51	57.2603914667
460	54	60.0521659632
480	56	62.8459069657
500	58	65.6371208084
520	61	68.4314042872
540	63	71.2239443758
560	66	74.0168967011
580	68	76.8105647955
600	71	79.6023937488
620	73	82.3970328148
640	75	85.1896613282

Table B.5: Bits of security for S and optimal values of v, for $q = 6$.

S	v	Bits of Security
100	8	7.30571083232
120	10	9.50421831578
140	12	11.7319149869
160	14	13.9822130489
180	16	16.2495594026
200	18	18.5296691895
210	19	19.6734717769
220	20	20.8193309744
230	21	21.9669726153
240	22	23.1161607396
260	24	25.4183920382
280	26	27.7247142078
300	28	30.0341520203
320	30	32.3459773081
340	32	34.6596442984
360	34	36.9747422712
380	36	39.2909608201
400	39	41.6088661402
420	41	43.927667836
440	43	46.2469409676
460	45	48.5665931552
480	47	50.8865532537
500	49	53.2067661194
520	51	55.5271887124
540	53	57.8477871864
560	55	60.1685347104
580	57	62.4894098314
600	59	64.8103952425
620	61	67.1314768514
640	63	69.4526430743
660	65	71.7738842981
680	67	74.0951924696
700	69	76.4165607799
720	71	78.7379834206
740	73	81.0594553947
760	75	83.380972369

Table B.6: Bits of security for S and optimal values of v, for $q = 7$.

Appendix C

Experiments Data

We include the data used to generate the graphs in Section 5.1.3. This data gives the running times for the algorithms in a scheme and the key and ciphertext sizes, for different sets of inputs. All times in the tables are in milliseconds, and averaged over 10 runs. All sizes in the tables are in bytes.

p	Bits	Setup	KeyGen	Encrypt	Decrypt
131	8	1.54998	0.0575747	3.973061	0.7194238
257	9	1.315528	0.0468375	3.558248	0.8388754
521	10	1.349419	0.046637	3.81149	0.989154
1,031	11	1.468602	0.0515353	4.547757	1.130014
2,053	12	1.654349	0.0543424	5.191734	1.315738
4,099	13	1.811205	0.0656911	6.421297	1.505172
8,209	14	2.22609	0.0785932	8.275854	1.710654
16,411	15	2.460622	0.0938597	8.258913	1.93491
32,771	16	2.957742	0.1005343	8.709874	2.129492
65,537	17	2.6628	0.0960014	10.113635	2.329285
131,101	18	2.363391	0.0998176	11.92076	2.543564
262,147	19	2.977107	0.1068233	11.86837	2.765922
524,309	20	3.142816	0.102723	11.408775	2.876541
1,048,583	21	3.298215	0.1304621	13.47584	3.081963
2,097,169	22	3.34881	0.1158222	14.10905	3.296371
4,194,319	23	3.729242	0.1254665	14.48263	3.536734
8,388,617	24	3.661746	0.1236207	15.78591	3.770024
16,777,259	25	3.594331	0.1279305	15.8602	4.116024
33,554,467	26	4.224878	0.1447126	18.32285	4.360517
67,108,879	27	4.182325	0.1401668	18.22762	4.539007
134,217,757	28	4.443883	0.1509604	19.04844	4.755402
268,435,459	29	4.855381	0.1489859	19.069	5.056261
536,870,923	30	4.570328	0.1600769	20.88948	5.318429
1,073,741,827	31	4.61533	0.1605114	21.02819	5.569664

Table C.1: One-key SS running times for different primes, with AES, inner product mod p circuits, for length 10 vectors.

		MSK/MPK		FK	CT
p	Bits	size	keys	size	size
131	8	2,964	160	1,527	1,802,437
257	9	3,334	180	1,717	2,245,862
521	10	3,704	200	1,907	2,737,982
1,031	11	4,074	220	2,097	3,278,769
2,053	12	4,444	240	2,287	3,882,973
4,099	13	4,814	260	2,477	4,600,470
8,209	14	5,184	280	2,667	5,355,466
16,411	15	5,554	300	2,857	6,164,132
32,771	16	5,924	320	3,047	7,026,308
65,537	17	6,294	340	3,237	7,942,268
131,101	18	6,664	360	3,427	8,911,894
262,147	19	7,034	380	3,617	9,934,916
524,309	20	7,404	400	3,807	11,011,798
1,048,583	21	7,774	420	3,997	12,142,352
2,097,169	22	8,144	440	4,187	13,326,178
4,194,319	23	8,514	460	4,377	14,563,938
8,388,617	24	8,884	480	4,567	15,855,268
16,777,259	25	9,254	500	4,757	17,200,172
33,554,467	26	9,624	520	4,947	18,598,500
67,108,879	27	9,994	540	5,137	20,050,568
134,217,757	28	10,364	560	5,327	21,556,272
268,435,459	29	10,734	580	5,517	23,115,586
536,870,923	30	11,104	600	5,707	24,728,404
1,073,741,827	31	11,474	620	5,897	26,395,026

Table C.2: One-key SS key and ciphertext sizes for different primes, with AES, inner product mod p circuits, for length 10 vectors.

Length	Setup	KeyGen	Encrypt	Decrypt
1	0.3231109	0.0171254	0.6708636	0.1746067
10	2.067322	0.0760288	6.340851	1.0319108
100	18.19843	0.5166602	37.82729	9.001745
200	35.48751	1.103201	74.2556	17.46849
300	52.73668	1.616318	109.3812	25.68324
400	68.69153	2.169415	146.9187	34.00203
500	84.4547	2.724473	182.5585	42.54663
600	101.51309	3.295232	219.8117	50.86044
700	119.3168	3.841888	255.4038	59.0062
800	134.4381	4.458252	292.9937	67.22722
900	150.4773	4.965566	329.7012	75.25928
1000	166.1338	5.601995	364.6297	84.24207
1100	180.5705	6.190041	397.4624	91.91721
1200	197.7791	6.775606	434.0631	100.65667
1300	213.0395	7.37893	470.9584	108.842
1400	226.997	7.911448	504.0103	117.2579
1500	244.4314	8.436557	541.1995	125.6439
1600	259.6933	9.004056	582.6269	133.9788
1700	278.3793	9.59012	613.0953	143.7887
1800	293.8813	10.26022	649.9419	151.8099
1900	311.357	10.80076	687.2063	160.6146
2000	325.8608	11.36094	724.762	168.8667
2200	360.9447	12.58547	807.7214	188.3495
2400	393.9359	13.85137	871.68	203.2647
2600	423.1235	14.90914	940.8234	219.5653
2800	457.8521	15.93526	1010.5493	236.0999
3000	499.2364	17.56474	1176.976	313.0198
3500	579.1119	20.57632	1280.37	378.9645
4000	662.6899	23.5416	1483.417	440.1461
4500	752.1038	26.26344	1660.821	544.7547
5000	808.9792	29.31875	1797.417	420.402
5500	883.9193	31.34131	1969.747	498.1895
6000	966.8491	33.69098	2145.656	534.0587
6500	1050.667	36.41234	2341.536	564.1144
7000	1125.326	39.51572	2494.641	614.0581
7500	1197.601	43.7912	2665.178	841.2505
8000	1268.571	45.26792	2851.259	879.0184
8500	1353.779	47.98199	3022.119	926.7474
9000	1430.911	50.57167	3216.517	1555.995
9500	1508.327	53.28335	3477.339	1673.2
10000	1624.991	57.13813	3577.108	1386.663

Table C.3: One-key SS running times for different length vectors, with AES, inner product mod 8123 circuits.

	MSK/MPK		FK	CT
Length	size	keys	size	size
1	483	26	250	100,732
10	4,814	260	24,77	1,036,937
100	48,104	2,600	24,707	12,938,677
200	96,204	5,200	49,407	26,243,739
300	144,304	7,800	74,107	39,548,961
400	192,404	10,400	98,807	52,854,045
500	240,504	13,000	123,507	66,159,247
600	288,604	15,600	148,207	79,464,309
700	336,704	18,200	172,907	92,769,541
800	384,804	20,800	197,607	106,074,625
900	432,904	23,400	222,307	119,379,845
1000	481,004	26,000	247,007	132,684,941
1100	529,104	28,600	271,707	145,990,151
1200	577,204	31,200	296,407	159,295,257
1300	625,304	33,800	321,107	172,600,457
1400	673,404	36,400	345,807	185,905,541
1500	721,504	39,000	370,507	199,210,761
1600	769,604	41,600	395,207	212,515,825
1700	817,704	44,200	419,907	225,821,021
1800	865,804	46,800	444,607	239,126,109
1900	913,904	49,400	469,307	252,431,307
2000	962,004	52,000	494,007	265,736,411
2200	1,058,204	57,200	543,407	292,346,717
2400	1,154,404	62,400	592,807	318,957,023
2600	1,250,604	67,600	642,207	350,491,537
2800	1,346,804	72,800	691,607	377,594,337
3000	1,443,004	78,000	741,007	404,697,137
3500	1,683,504	91,000	864,507	472,454,137
4000	1,924,004	104,000	988,007	540,211,137
4500	2,164,504	117,000	1,111,507	607,968,137
5000	2,405,004	130,000	1,235,007	675,725,137
5500	2,645,506	143,000	1,358,511	743,625,293
6000	2,886,006	156,000	1,482,011	811,538,293
6500	3,126,506	169,000	1,605,511	879,451,293
7000	3,367,006	182,000	1,729,011	947,364,293
7500	3,607,506	195,000	1,852,511	1,015,277,293
8000	3,848,006	208,000	1,976,011	1,083,190,293
8500	4,088,506	221,000	2,099,511	1,151,103,293
9000	4,329,006	234,000	2,223,011	1,219,016,293
9500	4,569,506	247,000	2,346,511	1,286,929,293
10000	4,810,006	260,000	2,470,011	1,354,842,293

Table C.4: One-key SS key and ciphertext sizes for different length vectors, with AES, inner product mod 8123 circuits.