Encrypted Computation

Tønnes Brekne
Department of Telematics
Norwegian University of Science and Technology

July, 2001

URN:NBN:no-2314

Abstract

The ability to construct software, call it a *functional ciphertext*, which can be remotely executed in encrypted form as an entirely self-contained unit, has the potential for some interesting applications. One such application is the construction of autonomous mobile agents capable of entering into certain types of legally binding contracts on behalf of the sender. At a premium in such circumstances is the ability to protect secret cryptographic keys or other secret information, which typically is necessary for legally binding contracts. Also important is the ability to do powerful computations, that are more than just one-off secure function evaluations.

The problem of constructing computation systems that achieve this, has been attempted by many to little or no avail. This thesis presents three similar cryptographic systems that take a step closer to making such encrypted software a reality.

First is demonstrated how one can construct mappings from finite automata, that through iteration can do computations. A stateless storage construction, called a Turing platform, is defined and it is shown that such a platform, in conjunction with a functional representation of a finite automaton, can perform Turing universal computation.

The univariate, multivariate, and parametric ciphers for the encryption of multivariate mappings are presented and cryptanalyzed. Cryptanalysis of these ciphers shows that they must be used very carefully, in order to resist cryptanalysis. Entirely new to cryptography is the ability to remotely and securely re-encrypt functional ciphertexts made with either univariate or multivariate encryption.

Lastly it is shown how the ciphers presented can be applied to the automaton representations in the form of mappings, to do general encrypted computation.

Note: many of the novel constructions in this thesis are covered by a patent application.

Contents

Foreword

This dissertation is delivered in partial fulfilment of the requirements of the Doktor Ingeniør degree the Norwegian University of Science and Technology. The work contained herein has been performed at the Department of Telematics, NTNU, Trondheim, under the supervision of Associate Professor Svein J. Knapskog.

The work has been supported by a joint grant from the Norwegian Research Council, Alcatel, Ericsson, Siemens, and Telenor.

Acknowledgements

This entire work was made possible by the academic freedom given me by my supervisor Svein J. Knapskog. Without it, and a broad interpretation of the research grant's area of application, this work might not have seen the light of day.

The entire basis of this thesis is due to a very unexpected breakthrough made late in the fall of 1998. This allowed me to significantly narrow my focus from my original, much more broadly defined thesis subject: securing autonomous mobile agents.

Chapter 1

Introduction

What is an encrypted computation? To explain this, a good starting point would perhaps be looking at a black box, where something comes in, and something else comes out. Assume that someone has to operate the box in order for it to process the input and produce output. If this someone can look inside the box, the process by which output is produced might be deduced. If so, the operator might modify the box to suit his or her purposes. The operator might also simply copy the box' design, and start selling it for profit. If the box' owner does not want this to occur, the box must be constructed so that it is practically impossible to deduce how it produces its output, even if the box is opened. As a part of such a design, the input or work space must be represented so as to be uninterpretable or unrecognizable within the box.

Imagine a process in place of the black box, which carries out any given computation on some host. The process and its work space may be regarded as plaintext, readable by any host upon which it executes. If such a process needs to keep one or more secrets, such as keys for generating digital signatures, as well as carry out a non-trivial computation using these secrets, it is necessary to make completely obscure the workings and work space of the process; particularly if the host is considered malicious. This obfuscation should preferably be done using a strong cryptosystem

of some sort. A computation carried out with such obfuscated code may be called an encrypted computation.

An example of such a process could be a mobile autonomous agent tasked with committing its sender to a purchase, after a sufficient amount of information is gathered to carry out the purchase. Such an agent is effectively a migrating process, that communicates with the hosts it visits. Each such host typically needs a support environment for such agents, which could be called the *host platform*. The agent is sent by some party, called the *sender*. An agent of this type can be of two types:

- the agent that is a true migrating process which carries along with it data, or
- an agent embedded in data, which is supposed to facilitate the use of the data by the host platform in some manner.

An autonomous mobile agent must be able to compute decision results, and act on them without communicating with its sender. If the agent needs to send a message to its host platform, where the contents have been derived using secret data (as when a digital signature is generated), those secret data must also be protected during storage and use. If the computation of a decision result depends on the use of secret data carried along with the agent's code, then those data must be protected in storage and during use. Furthermore, the result of the decision must still be useable by the agent.

There exist several systems for so-called secure multi-party computations. Many solve very specific multi-party computation problems. There is, however, a feature common to almost all of these systems: any given protocol completion can only compute at most a finite number of functions that require input of fixed size. They are therefore at best limited to computing in parallell several primitive recursive functions.

Related work has produced systems for evaluating primitively recursive functions. Furthermore, no prior work has been done on how to construct decision primitives that can be remotely evaluated and subsequently remotely used in a self-contained unit without directly revealing

the decision or the action taken because of the decision.

This dissertation presents a cryptographic system based on a special case of a problem proven to be NP-hard, which enables what I term encrypted computation. Encrypted computation is not to be mistaken for secure multi-party computations, which solve similar but not equivalent problems. The aim of the systems presented herein are to enable Turing-universal computation such that:

1. the program being executed is kept secret, even though it is executed on a platform where every aspect of the program's operation can be observed by a malicious entity;
2. the data manipulated by the secret program are kept secret, when the program dictates they be kept secret;
3. the program can read plaintext data if need be, without revealing the program itself or any secret data produced by it; and
4. the program can output plaintext data if need be, without revealing the program itself or any secret data still in use by it after the output.

Note in particular the third informal property; it effectively requires the program to be capable of encrypting plaintext data:

- without revealing key data used for the encryption; and preferably
- without revealing the encryption system used.

Similarly, the fourth property effectively requires the program to be capable of decrypting ciphertext data:

- without revealing key data used for the decryption; and preferably
- without revealing the decryption system used.

The dissertation presents three very similar systems with the aforementioned properties, each of which may be based on the symbolic composition of polynomials, and symbolic function composition using function tables. Cryptanalyses of the systems and their variants have been carried out. Lastly some open problems have been identified. An appendix

contains some composition algorithms for function tables and complexity analyses.

This was the informal introduction to the problems and solutions presented in this dissertation. The rest of this chapter will review some related work done by other researchers, before the problems are stated more formally.

1.1 Previous and Related Work

Related work has been done in the areas of privacy homomorphisms, zero-knowledge interactive protocols, secure multiparty computations, black box computations, copyright protection mechanisms, and hardware devices to support encrypted computation. I will now outline most of the relevant previous work in roughly chronological order. Formal details will be included where deemed relevant to illustrate the differences between previous work and the contents of this dissertation.

1.1.1 Privacy Homomorphisms

The ideas presented in this dissertation are based on a concept that may be traced back to the article [29] on privacy homomorphisms by Rivest, Adleman, and Dertouzos.

Let S be a set, and S' a possibly different set with the same cardinality as S. Let $D : S \longrightarrow S'$ be bijective. D is the decryption function. Denote an algebraic system for plaintext operations by

$$U = (S; f_1, \ldots, f_k; p_1, \ldots, p_l; s_1, \ldots, s_m),$$

where $f_i : S^{g_i} \longrightarrow S$ are functions with arity g_i, the p_i are predicates with arity h_i, and the s_i are distinct constants. Denote U's counterpart for computing with encrypted data by:

$$C = (S'; f'_1, \ldots, f'_k; p'_1, \ldots, p'_l; s'_1, \ldots, s'_m).$$

The mapping D is called a privacy homomorphism if it satisfies the following conditions:

1. $(\forall i)(1 \leq i \leq k \Rightarrow (\forall (a_1, \ldots, a_{g_i}) \in {S'}^{g_i})(\exists c \in S')$
   $(f'_i(a_1, \ldots, a_{g_i}) = c \Rightarrow f_i(D(a_1), \ldots, D(a_{g_i})) = D(c)))$

2. $(\forall i)(1 \leq i \leq l \Rightarrow (\forall (a_1, \ldots, a_{h_i}) \in {S'}^{h_i})$
   $(p'_i(a_1, \ldots, a_{h_i}) \Rightarrow p_i(D(a_1), \ldots, D(a_{h_i}))))$

3. $(\forall i)(1 \leq i \leq m \Rightarrow D(s'_i) = s_i)$

In order for C and D to be of any use as a protection, the following additional constraints should be satisfied:

1. D and D^{-1} are easy to compute.

2. The functions f'_i and predicates p'_i in C are efficiently computable.

3. D^{-1} is a non-expanding cipher or an expanding cipher whose cryptotext has a representation only marginally larger than the corresponding plaintext.

4. The operations and predicates in C should not be sufficient to yield an efficient computation of D.

Additionally, D^{-1} and D must resist ciphertext only and chosen plaintext attacks.

If a cryptographical system such as this could have existed, it would have been applicable for almost all problems which secure multiparty computations are designed to solve. This is not the case.

Rivest, et. al. [29] point out that no predicate that imposes a total order on S or S' is allowable, as this allows the efficient computation of D and D^{-1} using binary search and the predicate in question. A consequence of this is that decision primitives that make use of comparisons of the type "greater-than", "greater-than-or-equal-to", and so on, cannot securely be encrypted as predicates.

Furthermore, an analysis of some of the example homomorphisms presented in [29], by Brickell and Yacobi [13] demonstrates that two of the presented homomorphism schemes are vulnerable to cryptanalysis. Later

work by Boneh and Lipton in [11] (summarized below) shows that similar systems, based on algebraically homomorphic functions, are in general unattainable for finite fields or extensions of finite fields.

1.1.2 Encrypted Computation Using Microprocessors

There are several patented systems for executing encrypted machine code. Most of these systems are intended to solve one or more of the following problems:

- Alice has a program p that Bob wants to use. Alice does not want Bob to see any of p's contents, but p must be executable by Bob on his own computer. It should also be possible for p to use plaintext data supplied by Bob if necessary.

- Alice has a program p that Bob wants to use. The program p must be executable on Bob's computer, and *only* on Bob's computer, so that piracy becomes impractical.

Best describes in [5]–[9] different variants of "crypto-microprocessors" that execute encrypted machine code. Both the instructions and their operands are encrypted. These systems depend on the microprocessor having a block cipher on die to do decryption and encryption. Minor variations of these systems have been designed by Hampson [21] (the cipher is physically located outside the CPU), and Lumley [26] (how to construct a CPU that can execute both encrypted and unencrypted code seamlessly).

The basic principle of these systems is that instructions and operands fetched from the main memory are decrypted upon arrival in the CPU prior to processing in the instruction decoder. Similarly, all data being written from the CPU to the main memory is encrypted before it leaves the CPU. Best describes different ways of ensuring that more than one key is used in the encryption/decryption of a program and its workspace, such that usage of individual encryption keys is minimized. Among other things his inventions allow the use of addresses in the generation of keys for individual instruction blocks, partitioning of memory into zones with

different associated encryption keys, etc. These inventions manage to solve both the problems mentioned above. Bob can actually execute a program which without cryptanalytic work is uninterpretable to him. By hardwiring processor keys, Alice can also make sure that programs are only executable on Bob's CPU, and not any other CPU, unless that CPU has an identical hard-wired key.

There are some obvious cryptographic weaknesses in these systems, mainly having to do with the amount of ciphertext available to attackers, and the fact that hard-wired keys are never changed. Nevertheless, they might still be sufficient for several applications.

If, however, such a CPU were widespread, and the employed cryptosystem were sufficiently resistant to cryptanalysis, it might be a moot point, as reading the code would require power analysis or similar physical measurement during code execution. Thus these systems provide encrypted computation under the assumption that:

- the keys stored in the CPU never leave the CPU, and are unreadable, and
- the CPU has instructions that specify whether input/output is in plaintext or not.

1.1.3 Protocols for Secure Computations

A lot of work has been done on protocols for secure computations. One of the first constructions is by Andrew Yao [35]. Yao studies the case where m people want to compute $f(x_1, \ldots, x_m)$ under the following conditions:

1. each person P_i intially knows only x_i, and does not know the value of any x_j for $j \neq i$
2. f must be computed such that after the computation, person P_i still knows the exact value of only x_i, and does not know the value of any x_j for $j \neq i$

The solution to this problem, detailed in [35], computes functions on the form $f : X_1 \times \cdots \times X_m \longrightarrow V$. Thus the protocol as it stands is only capable of computing primitive recursive functions.

The protocol implicitly assumes that all parties are in possession of secure computing bases upon which they can carry out their computations. Therefore the protocol does not tackle the problem of computing on a malicious host. Also, the protocol itself "is" the computation, thus remote encrypted computation is not possible using this protocol.

Stuart Haber attempts to progress from work by Yao and others, and constructs a protocol presented in [20], which can be used to simulate interactive Turing machine computations. This protocol, however, still assumes that the users are operating within a trusted computing base, since the computation itself is carried out within a domain under control of the user.

1.1.4 Zero-Knowledge Simulation of Boolean Circuits

Brassard and Crepeau [12] present a method of simulating boolean circuits using zero-knowledge interactive protocols. The security of one version of the protocol depends on the Quadratic Residuosity Assumption (see [28], page 99).

It may be summed up as follows:

> Bob computes the encrypted value of a function $f : \{0,1\}^k \longrightarrow \{0,1\}$ from the encrypted bits of the bit vector used as input. This is done with the help of Alice using a ZKIP.

An important point is that Bob cannot by himself compute the encrypted evaluation from encrypted data supplied by Alice. This effectively means that this method is still far from enabling encrypted universal Turing computation.

1.1.5 Multiparty Computations Ensuring Privacy of Each Party's Input and Correctness of the Result

Chaum, Damgård, and van de Graaf [16] present an alternative to Yao's protocols. Their alternative requires less computation, but hinges on the Quadratic Residuosity Assumption. In any case, the computational capabilities of this protocol are limited to primitive recursive functions for any individual protocol execution. This protocol solves a problem similar to—but not equivalent to—that solved in this dissertation.

1.1.6 On Hiding Information From an Oracle

Abadi, Feigenbaum, and Kilian present in [2] a discussion on computing with encrypted data. The problem considered is summed up in the article's abstract as follows:

> Player A wishes to know the value $f(x)$ for some x but lacks the power to compute it. Player B has the power to compute f and is willing to send $f(y)$ to A if she sends him y, for any y.

The point is then that A encrypts x, sends $y = E(x)$ to B, who then computes $f(y)$, returns this result to A, who then infers $f(x)$ from $f(y)$.

If one by inferral means decryption, then Abadi et.al. are effectively handling a problem very similar to that posed by privacy homomorphisms. This is similar to—but not equivalent to—the problem discussed in this dissertation. This dissertation discusses the problem where A knows f and at most part of x, and needs B in order to compute $f(x)$. A, however, does not wish B to know how f is computed so f must somehow be encrypted and still be able to accept plaintext input and produce cryptotext output. There is a significant difference between the two problems.

A special case of the problem solved in [2] is studied in [4]. This and [1] add nothing of importance in the context of this dissertation.

1.1.7 Computing with Encrypted Functions

In more recent years, Sander and Tschudin presented in [31] and [32] two potential candidates for encrypted computation:

1. polynomials encrypted with a particular type of privacy homomorphism, and

2. rational function composition, where one rational function is used to encrypt another.

Only the first scheme, called non-interactive evaluation of encrypted functions, is detailed in their work. Sander and Tschudin present a simple protocol demonstrating how it could work. The description below is taken directly from their paper with an insubstantial modification.

1. Alice encrypts f

2. Alice creates a program $P(E(f))$ which implements $E(f)$

3. Alice sends $P(E(f))$ to Bob.

4. Bob executes $P(E(f))$ using x as argument.

5. Bob sends $P(E(f))(x)$ to Alice.

6. Alice decrypts $P(E(f))(x)$ to obtain $f(x)$.

The encryption itself is done using an additively homomorphic encryption scheme on a ring $\mathbb{Z}_n$.

The second scheme is hardly mentioned, and is also the one which most resembles that presented in this paper. There is a difference, however, in that Sander and Tschudin do not demonstrate how to achieve anything more than evaluation of primitive recursive functions with any of their presented schemes.

1.1.8 Non-Interactive Encrypted Computation for NC^1

In [33], Sander, Young, and Yung study the following problems:

> **Computing with Encrypted Functions (CEF):** Alice has a circuit C and Bob has an input x. Alice wants to learn $C(x)$. Bob should compute and learn nothing about Alice's circuit C except its size.
>
> **Symmetrically Secure CEF:** Alice has a circuit C and Bob has an input x. Alice wants to learn $C(x)$. Bob should compute and learn nothing about Alice's circuit C except its size. Alice should learn nothing about x except what the value $C(x)$ reveals.

They solve this problem for universal circuits in NC^1, in practice circuits with limited depth and a finite number of inputs and outputs.

Their technique makes use of what they call non-interactive inattentive evaluation to achieve secure, oblivious evaluation of the circuit. Inattentive evaluation bases itself on rearranging bits in vectors of bits that in themselves represent encodings of the actual input bits. This makes possible the use of probabilistic encryption of such encodings to make the computations effectively oblivious.

A disadvantage is the expansion of the sizes of outputs relative to the sizes of the inputs. This expansion has worst case $\mathcal{O}(8^d)$ (for the OR-function), where d is the depth of the circuit. This means that iterative application of a circuit is not feasible, as the required storage (and computational effort) will increase exponentially with each computation step. Thus applications where variations in input and/or the number of computation steps are natural, are hard to compute securely using this algorithm.

1.1.9 One-Round Secure Computation

In [14], Cachin, Camenisch, Kilian, and Müller present a protocol based on secure multiparty computations as described by Yao, and all-or-nothing-disclosure-of-secrets. The presented protocol is capable of handling inputs

of variable lengths in a single computation run, as opposed to other work summarized in this section. The computation itself is constructed as a recursive evaluation cascade of circuits from different host platforms visited during the run of a hypothesized autonomous mobile agent.

It is important to note that the protected code is not the agent's code, but that of the host platforms visited by the agent. Thus Cachin et. al. have, contrary to their claims, not secured mobile agent code, but rather that of the hosts visited by the agent. As a matter of fact, what they effectively have is a protocol for a series of secure computations between Alice and a series of hosts, where the functions in question may be publicly known, and only the inputs are secret. In such a context, the concept of a mobile agent is irrelevant, except as an unnecessarily complicated messaging platform.

Their protocol does not:

- support autonomous encrypted computation by the agent,
- allow use of encrypted results during the computation, or
- protect any of the agent's code.

1.1.10 Black Box Fields

Boneh and Lipton study in [11] algebraically homomorphic encryption schemes in the context of black box fields. A black box field is defined as a six-tuple (p, n, h, F, G, T) where:

- p is a prime;
- n is a positive integer representing encoding length;
- $h : \{0,1\}^n \longrightarrow \mathbb{F}_p$ surjectively maps every n-bit binary string to an element in $\mathbb{F}_p$;
- $F, G : \{0,1\}^n \times \{0,1\}^n \longrightarrow \{0,1\}^n$ are functions performing addition and multiplication such that $h(F(x,y)) = h(x) + h(y)$ and $h(G(x,y)) = h(x)h(y)$.

- $T : \{0,1\}^n \times \{0,1\}^n \longrightarrow \{\texttt{true}, \texttt{false}\}^n$ is a function testing the equality of two black-box elements, such that $T(x, y) = \texttt{true}$ iff $h(x) = h(y)$.

An n-bit binary string representing some element $x \in \mathbb{F}_p$ is written $[x]$ for the remainder of this subsection. Denote the map sending x to $[x]$ by $[]$.

PROBLEM 1 (BLACK BOX FIELD PROBLEM) *Let (p, n, h, F, G, T) be a black box field for some prime p. Find an algorithm $A^{F,G,T,[]}$ that, given p and oracles for F, G, T, $[]$, and an element $[\alpha] \in \mathbb{F}_p$ computes α explicitly. Formally $A^{F,G,T,[]}([\alpha]) = a$ where $a \equiv \alpha(\mod p)$.*

Boneh and Lipton present subexponential algorithms that solve this problem. In effect this corresponds to cryptanalysis of a generalization of privacy homomorphisms.

1.1.11 Solutions Depending On Specified System Conditions

All of the above attempted solutions are general in that they do not require any special hardware/software combination and/or use in order to work. There is also much work on execution of encrypted programs based on specific hardware/software configurations.

1.1.12 Mobile Code Protection

In chapter 5 in [25], Loureiro presents a scheme to do secure remote function evaluation, where Alice sends Bob a function f in encrypted form $E(f)$, which is then applied to an input x given by Bob. This is similar to the construction by Sander and Tschudin, but is done for functions generally on the form $f : \mathbb{Z}_2^m \longrightarrow \mathbb{Z}_2^n$. It also has similar capabilities to the system proposed in [32].

As with the system of Yao, Sander, and others, it can only compute a primitive recursive function without Alice' interaction. Loureiro also introduces a system for reusing encrypted results of such function evaluations, but they depend on part of the computation occurring within tamper resistant hardware.

1.1.13 Tamper-resistant Execution of Programs

Aucsmith and Graunke describe in [3] a system for tamper-resistant execution of programs that operate on secret data. There are certain similarities with the systems presented in this dissertation. There are also critical differences between the inventions in [3] and the systems presented in this dissertation. The system in [3] requires:

- the existence of at least one "programming instruction block" in plaintext,
- the existence of at least one "memory cell" in plaintext,
- the recovery of subsequent program instruction plaintexts in order for an execution to continue, and
- the recovery of the data in plaintext for subsequent program execution.

The system in effect gradually decrypts a program and its requisite data as the computation progresses. This is in other words not execution of an encrypted program, but merely a system for program obfuscation.

Glover describes in [19] how any digital information product can be protected by enclosing it in a program that does authentication, license checks etc. before decrypting any part of the product. The system describes by Glover does not actually achieve *encrypted* computation, as all code is successively decrypted by some plaintext code prior to execution. The system hinges on the assumption that the initialization code will execute in protected storage—an assumption which cannot be made for the purposes for this dissertation.

Maliszewski describes in [27] a system for storing and using secret data and secret execution code in memory. This system is similar to the system of tamper resistance by Aucsmith and Graunke in [3]. It decrypts and encrypts code data without necessarily revealing the "keys" needed. It does not, however, achieve true encrypted computation, as secrets stored in memory must be decrypted as they are used, or before they are changed.

1.2 The Problem

The problem consists of the following:

- Expressing programs in an encryptable form. This problem can be broken into:
  - expressing program code or state machines in an encryptable form, and
  - expressing program branching in an encryptable form.
- Encrypting program representations.
- Using encrypted program representations, which can be broken into:
  - constructing encrypted program representations such that program state can be held secret during execution,
  - constructing encrypted program representations such that encrypted programs can encrypt received plaintext input without revealing keys and decrypt ciphertext to plaintext output without revealing keys.

In the following, denote by I the input space, S the state space, and O the output space. A more formal statement of these problems could be as follows:

PROBLEM 2 (ENCRYPTABLE REPRESENTATION) *Given a class $\mathcal{M}$ of automata, does there exist a class of representations $\mathcal{F}$ and a transformation $T_F : \mathcal{M} \longrightarrow \mathcal{F}$, such that:*

1. *elements in $\mathcal{F}$ can be used directly in computation;*
2. *T_F can be efficiently computed; and*
3. *$T_F(M)$ is encryptable?*

PROBLEM 3 (PROGRAM ENCRYPTION) *Given a class of representations $\mathcal{F}$ satisfying property 1 in problem 2, does there exist a class of tranformations $E : K \times \mathcal{F} \longrightarrow \mathcal{F}$, where K is a class of keys, such that:*

1. *F can be encrypted in part or in its entirety;*
2. *E can be efficiently computed;*
3. *elements in K can be efficiently generated; and*
4. *$E(F)$, $F \in \mathcal{F}$, can still be used directly in computation such that:*
   (a) *one or more outputs of F may be encrypted;*
   (b) *one or more inputs of F may be encrypted; and*
   (c) *the state space/work space of F may be encrypted partially or completely?*

PROBLEM 4 ((TURING) UNIVERSAL ENCRYPTED COMPUTATION) *Does there exist at least one element $F \in \mathcal{F}$ such that $E(F)$ is capable of (Turing) universal encrypted computation, such that a storage of $E(F)$ bijectively mappable to a Turing machine's tape contains only encrypted values?*

PROBLEM 5 (STRONG PROGRAM ENCRYPTION) *Does there exist a class of transformations $E : K \times \mathcal{F} \longrightarrow \mathcal{F}$ satisfying problem 3 such that:*

1. *the encrypted portion of the program representation is strongly encrypted;*
2. *the encrypted portion of the state of the encrypted program is strongly encrypted;*
3. *encrypted output is strongly encrypted; and*
4. *encrypted input is strongly encrypted?*

PROBLEM 6 (STRONG (TURING) UNIVERSAL ENCRYPTED COMPUTATION) *Does there exist a class of transformations satisfying problem 5 and an element $F \in \mathcal{F}$ such that $E(F)$ also satisfies problem 4?*

Chapter 2

Encryptable Representations of Automata

This chapter details three different ways of representing finite state automata such that they are amenable to encryption. It also reviews table representations of finite state automata (FSA).

2.1 The Lagrange Interpolation of a Mealy Machine

A Mealy machine as defined in [22] is a six-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where Q is the set of states, Σ the input alphabet, Δ the output alphabet, $\delta : Q \times \Sigma \longrightarrow Q$ the state transition function, $\lambda : Q \times \Sigma \longrightarrow \Delta$ the output function, and q_0 the initial state. In some cases the state transition function and output function may not be defined for certain pairs $(q, \sigma) \in Q \times \Sigma$. To ensure generality, δ and λ are defined over a domain $D \subseteq Q \times \Sigma$.

Fix a Mealy machine M such that $Q \subseteq \mathbb{Z}_N^S$, $\Sigma \subseteq \mathbb{Z}_N^I$, and $\Delta \subseteq \mathbb{Z}_N^O$, and S, I, and O are fixed positive integers. N is a fixed prime number or a fixed power e of a prime number P. If N is a prime number, the elements of $\mathbb{Z}_N$ are interpreted as integers, and the addition operation is addition modulo N, while the multiplication operation is multiplication modulo N. If $N = P^e$ where P is a prime number, and e a positive integer, the elements

of $\mathbb{Z}_N$ are interpreted as polynomials over $\mathbb{Z}_P$, that are elements in $\mathbb{Z}_P[x]$ modulo a fixed irreducible polynomial in $\mathbb{Z}_P[x]$ of degree e. In such a case the addition operation is the addition of polynomials modulo $p(x)$. The multiplication operation is the multiplication of polynomials modulo $p(x)$. Elements in $\mathbb{Z}_P[x]/p(x)$ can be isomorphically mapped to $\mathbb{Z}_{P^e}$. $\mathbb{Z}_P[x]/p(x)$ is also a field, whose multiplicative subgroup has order $P^e - 1$.

Any Mealy machine's transitions (and output) are effectively defined by a set of quadruplets (*state*, *input*, *next-state*, *output*). Fix the domain of M as $D \subseteq Q \times \Sigma \subseteq \mathbb{Z}_N^S \times \mathbb{Z}_N^I$. It is possible to extend δ and λ to mappings

$$\tilde{\delta} : \mathbb{Z}_N^S \times \mathbb{Z}_N^I \longrightarrow \mathbb{Z}_N^S, \text{ and} \tag{2.1}$$
$$\tilde{\lambda} : \mathbb{Z}_N^S \times \mathbb{Z}_N^I \longrightarrow \mathbb{Z}_N^O. \tag{2.2}$$

Let $\vec{x} \in \mathbb{Z}_N^S$ denote M's state, and $\vec{y} \in \mathbb{Z}_N^I$ its input. The components of the mappings $\tilde{\delta}$ and $\tilde{\lambda}$ may be constructed as polynomials in $\mathbb{Z}_N[\vec{x}, \vec{y}] = \mathbb{Z}_N[x_1, \ldots, x_S, y_1, \ldots, y_I]$ as follows:

$$\tilde{\delta}(\vec{x}, \vec{y}) = \sum_{(\vec{i},\vec{j}) \in D} a_{i_1}(x_1) \cdots a_{i_S}(x_S) a_{j_1}(y_1) \cdots a_{j_I}(y_I) \delta(\vec{x}, \vec{y}), \text{ and} \tag{2.3}$$
$$\tilde{\lambda}(\vec{x}, \vec{y}) = \sum_{(\vec{i},\vec{j}) \in D} a_{i_1}(x_1) \cdots a_{i_S}(x_S) a_{j_1}(y_1) \cdots a_{j_I}(y_I) \lambda(\vec{x}, \vec{y}) \tag{2.4}$$

where

$$a_i(x) = \prod_{k \in \mathbb{Z}_N, k \neq i} \frac{x - k}{i - k}. \tag{2.5}$$

The functions $a_i(x)$ are Lagrange polynomials. For $i \in \mathbb{Z}_N$ $a_i(x) = 1$ iff $x = i$ and $a_i(x) = 0$ for $x \in \mathbb{Z}_N - \{i\}$.

$\tilde{\delta}$ and $\tilde{\lambda}$ are the Lagrange interpolations of δ and λ, respectively. This mapping is called the *computing endomorphism* of M. All the $|D|$ function values of δ are used in an interpolation over all $|D|$ distinct elements in the domain D. Thus $\tilde{\delta}$ is an *exact* interpolation of δ at all elements for which δ is defined; the points in D. Similarly, $\tilde{\lambda}$ is an exact interpolation of λ at all elements in D.

Denote by $\tilde{M}$ the equivalent machine given by the polynomial interpolations (and extensions) of the state transition/output functions defined for a Mealy machine M.

The mapping (δ, λ) is not defined for arguments outside D, while the polynomial mapping $(\tilde{\delta}, \tilde{\lambda})$ is. Thus $(\tilde{\delta}, \tilde{\lambda})$ allows the automaton to be run on arguments outside D even though (δ, λ) is not originally defined outside D. Any elements outside D (and thus in $\mathbb{Z}_N^S \times \mathbb{Z}_N^I - D \supseteq Q \times \Sigma - D$) may be defined for one or more of the following purposes:

1. ensure the "graceful" behaviour of M outside D,
2. make possible a particular mode of halting, and/or
3. increase uncertainty with respect to the original automaton, using obfuscation as a private random source.

Any such modification or combination of modifications of M is called *augmentation* in this document. Denote by M' a Mealy machine which may or may not have been augmented. Denote by $\delta', \lambda', \Delta', \Sigma', Q', D'$ the mappings and sets for M' corresponding to $\delta, \lambda, \Delta, \Sigma, Q, D$, respectively, for M. The symbol $\tilde{M}$ now denotes the polynomial mapping defined by symbolic interpolation of M'. Similarly, $\tilde{\delta}$ and $\tilde{\lambda}$ are henceforth the Lagrange interpolations of δ' and λ', respectively.

Assume that each element of $\mathbb{Z}_N$ requires $S_{s,N}$ units of storage. Under this assumption, the space complexity of storing the tabular representation of M' is at least $2|D'|S_{s,N}$, where D' is the domain of (δ', λ') (state transition and output functions of M'). Each computation step using the tabular definition of M' directly can be computed in the time it takes to do two table lookups.

The polynomial representation has a space complexity of at least $2|Q' \times \Sigma'|S_{s,N}$, as it is usually dense. If $|Q' \times \Sigma'| - |D'|$ is large, the polynomial may require significantly more storage than the tabular representation. Each computation step requires the evaluation of two polynomials with at least $|Q' \times \Sigma'|$ coefficients. Each evaluation requires at least $|Q' \times \Sigma'| - 1$ multiplications and additions, or more, depending on the exact representation

of Q' and Σ' and Δ' over $\mathbb{Z}_N$. Depending on the representation, a temporary workspace of a size polynomial in N may also be needed to speed evaluation.

2.2 Blum-Shub-Smale Automata

Blum-Shub-Smale automata (hereafter called BSS automata) are defined in [10]. They appear to be better suited for some implementation purposes in this dissertation's context than the more common finite automata representations. BSS automata are defined generally for rings and fields. The relevant variant is a BSS automaton over a finite field, primarily $\mathbb{Z}_N$ when N is a power of a prime number. Since the automata were originally designed for infinite rings, a restriction to a finite field $\mathbb{Z}_N$, and the introduction of a sort of "interactivity" necessitates some adjustment of the original automaton definition.

2.2.1 Definition of BSS-automata

Blum, Shub, and Smale define a theory of computation over rings in [10]. This theory encompasses both finite and infinite automata over some ring R. This subsection shows that finite BSS automata can be expressed using a single polynomial mapping, when R is a finite field of order N. As before, N is a power of a prime number. If N is a prime number, then the elements of the field $\mathbb{Z}_N$ are interpreted as integers modulo N. Otherwise, the elements of $\mathbb{Z}_N$ are interpreted as polynomials in $\mathbb{Z}_P[x]/p(x)$, where $N = P^e$, and $p(x)$ is a fixed irreducible polynomial over $\mathbb{Z}_P$.

DEFINITION 1 (BSS AUTOMATON OVER R) *A* Blum-Shub-Smale (BSS) automaton over R *with finite-dimensional state space consists of:*

- *an input space* $\overline{I} = R^I$,
- *an output space* $\overline{O} = R^O$,
- *a state space* $\overline{S} = R^S$, *and*

- *a directed graph with $\mathcal{N}$ numbered nodes,*

where I, O, S, and $\mathcal{N}$ are positive integers.

The graph of the automaton has four node variants, numbered by type in the list below:

1. *Input node (node of type 1)*
   This node has one outgoing edge to the node numbered $\beta(n)$ and no incoming edges. The number of the input node is n. Associated to this node is the injective input mapping $\tilde{I} : \overline{I} \longrightarrow \overline{S}$. There is only one input node in any automaton over R.

2. *Output node (node of type 2)*
   These nodes have one incoming edge, and no outgoing edges. The computation of the automaton is finished when an output node is reached. Each of these nodes has an output mapping $O_n : \overline{S} \longrightarrow \overline{O}$, where n is the number of the node in question.

3. *Computation node (node of type 3)*
   Each node of this type, numbered n, has one incoming and one outgoing edge to node number $\beta(n)$. To each such node is associated a map $g_n : \overline{S} \longrightarrow \overline{S}$. g_n is rational for R a field, and polynomial otherwise.

4. *Branch node (node of type 4)*
   Each node number n of this type has one incoming edge, and two *outgoing edges to the nodes numbered $\beta^-(n)$ and $\beta^+(n)$. To each such node is associated a polynomial or rational (for R a field) map $h_n : \overline{S} \longrightarrow R$. If R is an ordered ring, the automaton "moves" to node $\beta^-(n)$ when $h_n(\vec{x}) < 0$, $\vec{x} \in \overline{S}$, and to node number $\beta^+(n)$ when $h_n(\vec{x}) \geq 0$. If R is not ordered, the automaton "moves" by convention to node $\beta^-(n)$ when $h_n(\vec{x}) = 0$ and to node number $\beta^+(n)$ when $h_n(\vec{x}) \neq 0$.*

The analogue of the classical finite automaton has now been defined. The automaton defined above has a bounded "memory"—a state vector

with finitely many components. It would be nice to accommodate infinite automata, but such an automaton is not expressible as a polynomial with finite dimension and a finite number of monomials.

The *dimension* of an automaton M is defined as

$$\dim M = \max_n \{\dim g_n, \dim h_n\},$$

where $n \in \{1, \ldots, \mathcal{N}\}$ and $\mathcal{N}$ is the number of nodes in the automaton's graph. The *degree* of an automaton M is defined as

$$\deg M = \max_n \{\deg g_n, \deg h_n\}.$$

Since only finite-dimensional BSS automata are considered, it is possible to simplify some of the notation used in [10], and let x_k refer to the k^{th} component in the state vector.

2.2.2 The Computing Endomorphism for BSS Automata

For each "move", the automaton executes the following two steps:

1. Compute new state: $\vec{x} \mapsto g_n(\vec{x})$.
2. Change "location" from node number n to the next node, node number $\beta(n)$, or one of $\beta^+(n)$, $\beta^-(n)$ for n a branch node.

A complete and precise description of the state of the automaton may be written

$$(n, \vec{x}) \in \overline{N} \times \overline{S},$$

where $\overline{N} = \{1, \ldots, \mathcal{N}\}$ is the set of nodes in the automaton's graph, and $\overline{S}$ is the state space containing the automaton's "registers". The space $\overline{N} \times \overline{S}$ is called the *full state space*.

Instead of considering the state transition function, defined explicitly as with classical finite automata, the automaton's *computing endomorphism*

$$H : \overline{N} \times \overline{S} \longrightarrow \overline{N} \times \overline{S}$$

is considered. The computing endomorphism is in general defined as

$$H(n, \vec{x}) = (\beta(n, \chi(\vec{x})), g_n(\vec{x})),$$

where β is the *next node function*, computing the node the automaton is to "move" to when g_n has been applied to the state vector $\vec{x}$. The sign function, denoted by $\chi(\vec{x})$, is defined as follows:

$$\chi(\vec{x}) = \begin{cases} 1, & x_1 > 0 \\ 0, & x_1 = 0 \\ -1, & x_1 < 0 \end{cases} \tag{2.6}$$

The next node function $\beta(n, \sigma) : \overline{N} \times \{-1, 0, 1\} \longrightarrow \overline{N}$ is in general

$$\beta(n, \sigma) = \begin{cases} \beta(n), & n < \mathcal{N} \text{ and } n \text{ is not a branch node} \\ \beta^+(n), & n \text{ is a branch node and } \sigma = 0, 1 \\ \beta^-(n), & n \text{ is a branch node and } \sigma = -1 \end{cases} \tag{2.7}$$

One may optionally require β to satisfy $\beta(\mathcal{N}) = \mathcal{N}$ as a convention. For some types of encrypted automaton execution, such convention may be necessary.

The computing endomorphism is a composition of the sign function χ and polynomial maps or rational maps. Fix an automaton M over $\mathbb{Z}_N$. Let $B = \{\text{branch nodes in } M\}$, and for BSS automata define

$$a_n(y) = \prod_{j \neq n, j \in \overline{N}} \frac{(y - j)}{(n - j)}. \tag{2.8}$$

When $y \in \overline{N}$, $a_n(y) = 1$ iff $n = y$ and $a_n(y) = 0$ otherwise. For $y \notin \overline{N}$, $a_n(y)$ produces nonsense. Next express $\beta(y, \sigma) = \beta(y, \chi(\vec{x}))$ as the polynomial below

$$\beta(y, \sigma) = \sum_{n \in \overline{N} - B} a_n(y)\beta(n) + \frac{1}{2}\left(-\sigma^2 + \sigma + 2\right) \sum_{n \in B} a_n(y)\beta^+(n) + \frac{(\sigma^2 - \sigma)}{2} \sum_{n \in B} a_n(y)\beta^-(n). \tag{2.9}$$

The first sum in equation 2.9 computes the next node for any node y that is not a branch node. The last two sums compute the next node for all branch nodes y. Note that equation 2.9 gives a polynomial expression for β when y and σ are given. When computing $\beta(y, \sigma)$ for a node, $\sigma = \chi(\vec{x})$ must be evaluated. Over a finite field it is possible to express χ as a polynomial.

A mapping $g(n, \vec{x}) = g_n(\vec{x})$ does all "useful" computation in M. Let

$$g(y, \vec{x}) = \sum_{n \in \overline{N}} a_n(y) g_n(\vec{x}),$$

where $a_n(y)$ is defined in equation 2.8. One may in general write

$$g_n(\vec{x}) = \left(\frac{f_{n,1}(\vec{x})}{q_{n,1}(\vec{x})}, \frac{f_{n,2}(\vec{x})}{q_{n,2}(\vec{x})}, \frac{f_{n,3}(\vec{x})}{q_{n,3}(\vec{x})}, \dots \right),$$

where $f_{n,l}(\vec{x})$ and $q_{n,l}(\vec{x})$ are polynomials in general. Since all mappings considered here are over a finite field, $q_{n,l} \equiv 1$ for all l. If n is a computation node, $f_{n,l}$ is a polynomial in $\vec{x}$ with dimension bounded by $\dim M$, and degree bounded by $\deg M$. If n is *not* a computation node, then $q_{n,l}(\vec{x}) \equiv 1$ and $f_{n,l}(\vec{x})$ is identical to the l^{th} component of $\vec{x}$ for all l. It is then possible to express $g(n, \vec{x})$ like this:

$$g(n, \vec{x}) = \left(\frac{\sum_{n \in \overline{N}} a_n(y) f_{n,1}(\vec{x})}{\sum_{n \in \overline{N}} a_n(y) q_{n,1}(\vec{x})}, \frac{\sum_{n \in \overline{N}} a_n(y) f_{n,2}(\vec{x})}{\sum_{n \in \overline{N}} a_n(y) q_{n,2}(\vec{x})}, \dots \right) \tag{2.10}$$

This gives the explicit expression for the computing endomorphism for M on the form

$$H(n, \vec{x}) = (\beta(n, \chi(\vec{x})), g(n, \vec{x})). \tag{2.11}$$

At this stage, H is at best piecewise polynomial. The next section introduces some adaptions, enabling the expression of H with one polynomial mapping. The result of the adaption will be called a BSS' automaton.

2.2.3 Adapting BSS Automata to a Finite Field

Denote by $\mathcal{N}$ the number of nodes in the automaton, and by d the dimension of the automaton. Set $\overline{N} = \mathbb{Z}_{\mathcal{N}}$, and $\overline{S} = \mathbb{Z}_N^d$, so that the computing

endomorphism H is a mapping

$$H : \mathbb{Z}_{\mathcal{N}} \times \mathbb{Z}_N^d \longrightarrow \mathbb{Z}_N \times \mathbb{Z}_N^d.$$

Note that the first node will hereafter be numbered "0". For each node $n \in \mathbb{Z}_{\mathcal{N}}$, there is a mapping $g_n : \mathbb{Z}_N^d \longrightarrow \mathbb{Z}_N^d$, which for N a prime power is in general componentwise rational. The following restrictions are introduced:

1. g_n may only be polynomial.

2. Every node may do branching, and computation. One or more nodes may be designated "halting" nodes.

3. The dimension of each automaton is the dimension of its state-space $\overline{S}$, which is d. If $d > \dim\overline{S}$, then restrict $\overline{S}$ such that $d = \dim\overline{S}$.

Since these automata are constructed over the finite field $\mathbb{Z}_N$, which contains only non-negative integers, the original branch node given in [10] becomes meaningless. Instead, the next-node function should have the form $\beta : \overline{N} \times \mathbb{Z}_N \longrightarrow \overline{N}$. To simplify, require $\overline{N} \subseteq \mathbb{Z}_N$, even though one could make do with a smaller prime than some $N \geq \mathcal{N}$ for the state-space. Thus $\beta \in \mathbb{Z}_N[x_1, x_2]$.

Since the relation ≥ 0 is trivial over $\mathbb{Z}_N$, it is necessary with the more general set membership relation $\in K$, where $K \subseteq (\mathbb{Z}_N - \{0\}) = \mathbb{Z}_N^*$, the multiplicative subgroup of $\mathbb{Z}_N$. For $K \subseteq \mathbb{Z}_N^*$, define

$$b_K(z) = \prod_{i \in \mathbb{Z}_N^* - K} (z - i)^{N-1}. \tag{2.12}$$

When $z \in \mathbb{Z}_N$, $b_K(z)$ maps to 1 iff $z \in K$ and to 0 otherwise. The function b_K exploits a property of elements of the finite multiplicative subgroup $\mathbb{Z}_N^*$ of the finite field $\mathbb{Z}_N$, which effectively implies $x^{N-1} \equiv 1(\mod N)$. Since 0 is not in this subgroup, it does not satisfy this property, and thus cannot be included in K.

Let $B \subset \mathbb{Z}_{\mathcal{N}}$ be the set of all branch nodes. Using b_K, it becomes possible to express β using a polynomial:

$$\beta(n,x) = \sum_{i=0}^{N-1} a_i(n)\Delta(i,x), \tag{2.13}$$

where

$$\Delta(i,x) = \begin{cases} n_i', & i \notin B \\ b_{K_i}(x)n_1' + (1 - b_{K_i}(x))\, n_2', & i \in B. \end{cases} \tag{2.14}$$

The constants n_i', n_1', and n_2' are all elements in $\mathbb{Z}_{\mathcal{N}}$, the node space. This enables the expression of the computing endomorphism of BSS automata as a polynomial over $\mathbb{Z}_N$.

It is possible, however, to generalize the next-node function. For each $i \in B$, define $K_i = \cup_{j=1}^{k_i} K_{i,j}$ such that for a fixed i all $K_{i,j}$ are mutually disjoint. Each set $K_{i,j}$ thus has its own $b_{K_{i,j}}$ defined in the same way as b_K was in equation 2.12. This allows the definition of up to $\mathcal{N}$ distinct branches to nodes $n_{i,j}$ from any given node. The resulting expession for $\Delta(i,x)$ is then:

$$\Delta(i,x) = \begin{cases} n', & i \notin B \\ \sum_{j=1}^{k_i} b_{K_{i,j}}(x)n_{i,j} + (1 - b_{K_i}(x))\, n_{i,k_i+1}, & i \in B. \end{cases} \tag{2.15}$$

The constants n', all n_{i,k_i+1}, and all $n_{i,j}$ are elements in $\mathbb{Z}_{\mathcal{N}}$, the node space. This enables the expression of the computing endomorphism of BSS automata as a polynomial over $\mathbb{Z}_N$.

Thus the computing endomorphism for BSS′ automata over $\mathbb{Z}_N$ is:

$$H\left(n,\vec{x}\right) = \left(\beta\left(n,x_C\right), \sum_{i=0}^{\mathcal{N}-1} a_i(n)g_i\left(\vec{x}\right)\right), \tag{2.16}$$

where $1 \leq C \leq S+1$ is fixed.

As a convention, every halting node n is defined such that:

- the complete automaton state remains unchanged (including node number) $\beta(n,x) = n$, and

- a constant "blank" symbol is output irrespective of input.

The resulting definition of a BSS' automaton is then:

DEFINITION 2 (BSS' AUTOMATON) *An automaton over a selected finite field $\mathbb{Z}_N$ consisting of:*

- *an input space $\overline{I} = \mathbb{Z}_N^I$,*
- *an output space $\overline{O} = \mathbb{Z}_N^O$,*
- *a state space $\overline{S} = \mathbb{Z}_N^S$, and*
- *a directed graph with $\mathcal{N}$ numbered nodes,*

where I, O, S, and $\mathcal{N}$ are positive integers.
Each node in the graph has:

- *a computation mapping $g_n : \mathbb{Z}_N^S \times \mathbb{Z}_N^I \longrightarrow \mathbb{Z}_N^S \times \mathbb{Z}_N^O$ which operates on state and input;*
- *a next node function $\beta : \mathbb{Z}_{\mathcal{N}} \times \mathbb{Z}_N \longrightarrow \mathbb{Z}_{\mathcal{N}}$ which computes the next node where computation will take place.*

Lastly, a component number $C \in \{2, \ldots, S+1\}$ of the state vector is selected for use as the second argument of β.

Because the automaton is defined over a finite field, and it is the evaluation of the computing homomorphism which is important, the resulting polynomial expression can be reduced such that it has degree no greater than $N-1$ in any variable. Thus space complexity for representation is bounded by $(S+O+1)N^{(S+I+1)}$. Time complexity for evaluation, assuming multiplication is the significant operation, is also bounded by $(S+O+1)N^{(S+I+1)}$.

2.3 A Register Automaton

The final representation is a representation specially tailored for parametric encryption as presented in section 4.8. Fix an $N = P^e$ such that P is a prime number, and e a positive integer. If $e = 1$, the elements of $\mathbb{Z}_N$ are interpreted as integers modulo N. If $e > 1$, the elements of $\mathbb{Z}_N$ are interpreted as polynomials in $\mathbb{Z}_P[x]/p(x)$, where $p(x)$ is a degree e fixed irreducible polynomial over $\mathbb{Z}_P$. Fix also positive integers d and m.

DEFINITION 3 (REGISTER AUTOMATON) *A* register automaton *is a tuple* $(\mathcal{P}, S, \vec{C}, \vec{D}, \mathcal{R}, f, g, h, q, N, d, m, \vec{H}, \vec{T})$*, where*

1. $d, m, N > 1$ *are positive integers;*
2. $\mathcal{P} = \{\vec{\mathcal{P}}_{\vec{\imath}}\}_{\vec{\imath} \in \mathcal{I}}$*, where* $\mathcal{I} \subseteq \mathbb{Z}_N^d$ *and* $\vec{\mathcal{P}}_{\vec{\imath}} \in \mathbb{Z}_N^d$ *is a program in the form of a series of instructions;*
3. $\vec{H}$ *is either an index vector in* $\mathbb{Z}_N^d$ *such that* $\exists \vec{\mathcal{P}}_{\vec{S}} \in \mathcal{P}$*, or not an element of* $\mathbb{Z}_N^d$*;*
4. $\vec{T}$ *is either a halting instruction in* $\mathbb{Z}_N^d$ *such that at least one element in* $\mathcal{P}$ *equals* $\vec{T}$*, or not an element of* $\mathbb{Z}_N^d$*;*
5. $\vec{C} \in \mathbb{Z}_N^d$ *is an instruction pointer;*
6. $S = \{\vec{S}_{\vec{\imath}}\}$ *is a set of vectors in* $\mathbb{Z}_N^d$ *indexed by vectors in* $\mathbb{Z}_N^d$*, where each* $\vec{S}_{\vec{\imath}}$ *is a storage cell;*
7. $\vec{D} \in \mathbb{Z}_N^d$ *is a storage pointer;*
8. $\mathcal{R} = (\vec{R}_1, \ldots, \vec{R}_m)$*, each* $\vec{R}_i \in \mathbb{Z}_N^d$*, are registers;*
9. *the next instruction pointer mapping is*

$$f(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}) : \mathbb{Z}_N^{d(m+4)} \longrightarrow \mathbb{Z}_N^d;$$

10. *the next storage pointer mapping is*

$$g(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}) : \mathbb{Z}_N^{d(m+4)} \longrightarrow \mathbb{Z}_N^d;$$

11. *the register transition mapping is*

$$h(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}) : \mathbb{Z}_N^{d(m+4)} \longrightarrow \mathbb{Z}_N^{kd},$$

where k is the number of registers not accepting input from the host platform, $0 \leq k \leq m$; and

12. *the storage transition mapping is*

$$q(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}) : \mathbb{Z}_N^{d(m+4)} \longrightarrow \mathbb{Z}_N^d.$$

This type of register automaton can accept input from its host platform in one or more of the following ways:

- through one or more registers,
- through one or more selected storage cells in the storage space,
- through the initial contents of the storage space, and
- through the initial contents of the instruction vectors.

In the case where one or more registers are used to accept input, the register transition mapping is adjusted so that it does not alter the contents of the registers accepting input from the host platform. The register automaton may also come with a list of registers and storage locations that function as outputs to the host platform.

A computation with this type of register automaton is initialized with the following steps:

- The initial values of $\vec{R}_1, \ldots, \vec{R}_m, \vec{C}, \vec{D}$ are given. Initial values for one or more storage cells $\vec{S}_{\vec{D}} \in S$ may also be given.

- All the elements in $\mathcal{P}$ are given.
- $\vec{\mathcal{P}}_{\vec{C}}$ and $\vec{S}_{\vec{D}}$ are given.

The computation step of this type of register automaton consists of the following steps:

1. Compute the next instruction pointer:
$$\vec{C}' = f(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}).$$
2. Compute the next storage pointer:
$$\vec{D}' = g(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}).$$
3. Compute the value to be written to the current storage cell:
$$\vec{S}' = q(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}).$$
4. Compute the register transition mapping:
$$\left(\vec{R}_{j_1}, \ldots, \vec{R}_{j_k}\right) = h\left(\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}\right).$$
5. Set $\vec{S}_{\vec{D}} = \vec{S}'$, $\vec{C} = \vec{C}'$, and $\vec{D} = \vec{D}'$.
6. Compute $\vec{\mathcal{P}}_{\vec{C}}$ and $\vec{S}_{\vec{D}}$.

The computation is considered to be ended when either $\vec{C} = \vec{H}$ or when $\vec{\mathcal{P}}_{\vec{C}} = \vec{T}$, or both, or a predefined value is written to a selected, fixed register or storage cell.

This register automaton may be implemented using either a polynomial representation, or a function table representation. Thus the mappings f, g, q, and h are defined either by polynomials or by function tables.

During use, there are no requirements as to when the host platform changes registers accepting input, and no requirements as to when the

host platform reads from designated "output" registers/storage cells. This allows computational work to be minimized.

Denote by $\vec{x}$ the vector consisting of the vectors $\vec{R}_1, \ldots, \vec{R}_m, \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}$ in that order. The closest one can get to a computing endomorphism for a register automaton is the mapping:

$$(h(\vec{x}), \vec{\mathcal{P}}_{f(\vec{x})}, \vec{S}_{g(\vec{x})}, f(\vec{x}), g(\vec{x})), \tag{2.17}$$

which doesn't take into account changes in storage cells.

2.4 A Tabular Representation

Mealy machines, BSS' automata, and the register automata defined in this dissertation may all be represented using polynomials or function tables. The polynomial representations may only be used when $N = P^e$, where P is a prime number, and e is a positive integer. In the case of BSS' automata, any valid such automaton must have a corresponding polynomial representation as described in section 2.2.3. The function table representations may be used for any integer $N \geq 2$.

In general the function table representation for $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ associates with each vector $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$ a vector $(f_1, \ldots, f_n) \in \mathbb{Z}_N^n$. For such a function table to be encryptable, the mapping values of f must be fully defined for all $\vec{x} \in \mathbb{Z}_N^m$. Thus an such table stores nN^m elements of $\mathbb{Z}_N$, but time complexity for evaluation is that of one table lookup, and so is $\mathcal{O}(1)$.

Chapter 3

Interactivity for Encryptable Automata

All the automata presented in chapter 2 are finite state automata. They have a finite control that produces output from an internal state and input. Mealy machines produce output with each computation step, while BSS' automata, as currently defined, produce output only when the computation is finished. Register automata are capable of both synchronous and asynchronous output and input in a way different from Mealy and BSS' automata.

What is desired is a single mapping, a computing endomorphism, that can repeatedly be applied to state and input to produce a new state and output for each computation step. To support a sufficiently general model of encrypted computation, automata must be able to communicate with other automata and oracles in their environment. This chapter introduces the necessary constructions to achieve this.

In general, M's computation step may be expressed using a mapping

$$f : \mathbb{Z}_N^S \times \mathbb{Z}_N^I \longrightarrow \mathbb{Z}_N^S \times \mathbb{Z}_N^O, \tag{3.1}$$

where S is the dimension of the internal state space, O the dimension of the output space, and I the dimension of the input space. The full state

space of M is $\mathbb{Z}_N^S \times \mathbb{Z}_N^O \times \mathbb{Z}_N^I$. Let $\vec{x}(i)$, $\vec{y}(i)$, and $\vec{z}(i)$ denote the state, input, and output, respectively, of M after i completed computation steps. These may be written together as follows

$$(\vec{x}(i), \vec{z}(i), \vec{y}(i)) = (x_1(i), \ldots, x_S(i), z_1(i), \ldots, z_O(i), y_1(i), \ldots, y_I(i)), \quad (3.2)$$

and may be regarded as a snapshot of M's execution. Note that $i \geq 0$. As a convention, f does not use already computed output for further computation. Furthermore, f does not alter $\vec{y}(i)$, as it is given for each computation step.

The initial state of M may be given as a vector $(\vec{x}(0), \vec{z}(0), \vec{y}(0)) \in \mathbb{Z}_N^S \times \mathbb{Z}_N^O \times \mathbb{Z}_N^I = \mathbb{S}$, where the contents of $\vec{z}(0)$ are unimportant with respect to the initialization of the automaton. Given M's state after n state-transitions, $\vec{x}(n)$, and the $(n+1)^{\text{st}}$ input $\vec{y}(n)$, the next state transition and output is computed by the mapping:

$$(\underbrace{f_1(\vec{x}(n), \vec{y}(n)), \ldots, f_S(\vec{x}(n), \vec{y}(n))}_{\text{next state}}, \underbrace{f_{S+1}(\vec{x}(n), \vec{y}(n)), \ldots, f_{S+I}(\vec{x}(n), \vec{y}(n))}_{\text{output}}, \underbrace{\vec{y}(n+1)}_{\text{next input}}). \quad (3.3)$$

For a Mealy machine, such a mapping would look like the following:

$$(\underbrace{\delta_1(\vec{x}(n), \vec{y}(n)), \ldots, \delta_S(\vec{x}(n), \vec{y}(n))}_{\text{next state}}, \underbrace{\lambda_1(\vec{x}(n), \vec{y}(n)), \ldots, \lambda_I(\vec{x}(n), \vec{y}(n))}_{\text{output}}, \underbrace{\vec{y}(n+1)}_{\text{next input}}). \quad (3.4)$$

The computation is executed by iteratively applying the mapping given in

equation 3.3. This gives the relations:

$$\vec{x}(n) = \begin{cases} (f_1(\vec{x}(n-1), \vec{y}(n-1)), \ldots, \\ \quad f_S(\vec{x}(n-1), \vec{y}(n-1))), & \text{for } n > 0 \\ \text{given for } n = 0 \end{cases} \tag{3.5}$$

$$\vec{z}(n) = \begin{cases} (f_{S+1}(\vec{x}(n-1), \vec{y}(n-1)), \ldots, \\ \quad f_{S+I}(\vec{x}(n-1), \vec{y}(n-1))), & \text{for } n > 0 \\ \text{given for } n = 0 \end{cases} \tag{3.6}$$

$$\vec{y}(n) \qquad \text{is given for } n \geq 0. \tag{3.7}$$

3.1 Turing Platform

In order to explicity demonstrate that the introduction of interactivity allows universal Turing computation (and later on: encrypted universal Turing computation), the Turing platform is introduced.

Informally, a *Turing platform* is storage device T consisting of an infinite tape of storage cells and a stateless finite control. When coupled to an appropriate finite state automaton, the result should be a Turing-like automaton, such as that described in appendix B.1. In general, however, it may be coupled to any oracle. A Turing platform performs computation steps in order to function as a storage device. Each such computation step consists of the following actions:

1. read the contents γ of the cell at the finite control;
2. write γ to the input of an oracle;
3. read the symbol α from the output of an oracle;
4. write α to the cell at the finite control; and
5. read the symbol d from the output of an oracle, and move left, right, or stand still, if possible, depending on the value of d.

A complete computation step for an automaton M interacting with a Turing platform T thus consists of the following actions:

1. T reads the contents γ of the cell at which its finite control is placed;
2. T writes γ to M's input;
3. M computes the next state;
4. M computes the output, including the direction of movement;
5. T reads the symbol α from M's output;
6. T writes α to the cell at the finite control;
7. T reads the symbol d from the output of an oracle; and
8. T moves left, right, or stands still, if possible, depending on the value of d.

DEFINITION 4 (TURING PLATFORM) *A* Turing platform *is a storage device* $T = (\Gamma, C, D, W)$*, where* Γ *is the tape alphabet, and* C*,* D*, and* W *are mappings defined below. The behavior of a Turing platform is defined by the mappings*

$C : \mathbb{N} \times \mathbb{N} \longrightarrow \Gamma$*, where* C*'s arguments are the tape position, and the number of completed computation steps, and* $C(p, i)$ *is the content of the tape cell at position,* p *after* i *computation steps. All* $C(p, 0)$ *are given.*

$P : \mathbb{N} \longrightarrow \mathbb{N}$*,* $P(0) = 0$*, where* P*'s argument is the number of completed computation steps, and* $P(i)$ *is the position of the finite control after* i *computation steps.*

$D : \mathbb{N} - \{0\} \longrightarrow \{-1, 0, 1\}$*, gives the direction in which the tape head moves in the* i^{th} *computation step: -1 is left, 0 is no move, 1 is right. If* $D(i) = 0$ *and* $P(i) = 0$*, then there is no defined next move as for Turing machines.*

$W : \mathbb{N} \longrightarrow \Gamma$*,* $W(i)$ *is the output to be written to cell number* $P(i)$ *in the* i^{th} *computation step.*

The index i keeps track of the completed computation steps, and is incremented only after every action of a computation step has been completed. The mappings $W(i)$ and $D(i)$ are given. The mappings must satisfy

$$\begin{array}{ll}(\forall i) & (i > 0 \rightarrow \\ & (P(i+1) = P(i) + D(i)) \wedge \\ & (C(P(i), i+1) = W(i)))\end{array} \tag{3.8}$$

The platform T represents the storage facilities offered M by the host. The interaction between M and T may be formalized by assigning:

1. $W(i)$ the value of a mapping applied to $\vec{z}(i+1)$ (output of M);
2. $D(i)$ the value of a mapping applied to $\vec{z}(i+1)$; and
3. assigning part (or all of) $\vec{y}(i+1)$ the value of a mapping of $C(P(i), i)$.

In the following, denote by $A|_B$ the projection of set A onto set B.

3.2 Halting Ms Computation

One or more methods of halting a computation may be desirable. There are at least five ways M's computation can be halted:

1. The classical way: M is fed an input of finite length and the computation halts when M's computing endomorphism $\tilde{M}$ has been applied to the last input $\vec{y}(n_{\text{final}})$. BSS' automata are initially not constructed for this type of halting.
2. Explicit halting symbol from the host: One or more input symbols is designated as a halting signal. The definition of such symbols accompanies the definition of M. The host stops executing M after $\tilde{M}$ has been applied to one such "signal".
3. Explicit halting symbol from M: One or more plaintext output symbols is defined as a halting signal. The definition of such symbols accompanies $\tilde{M}$. The host does not apply $\tilde{M}$ any further after reading such a symbol in the output.

4. Explicit halting state: One or more states/nodes are designated halting states/nodes. The definition of these states/nodes accompanies $\tilde{M}$. The host does not apply $\tilde{M}$ once one of these states/nodes is entered.

5. Detection of 1-cycles: The computing endomorphism is applied to the input until $\vec{x}(n+1) = \vec{x}(n)$ and $\vec{z}(n+1) = \vec{z}(n+1)$ and $\vec{y}(n+1) \neq \vec{y}(n)$.

More than one of the methods can be available for the same computation. Method 1 above imposes no intrinsic cost in time or space. It is possible to have interaction with the host platform prior to halting, which can affect the length of the input presented to the automaton.

Method 2 requires the definition of at least $|Q|$ (or $\mathcal{N}$ additional branches) entries, so that the halting signal can be received in any state. Halting requires one computation step. Unfortunately, the host has no explicit way of knowing whether the computation has actually halted in a proper manner or not.

Method 3 requires the definition of at least one part of one entry in the automaton's output function, that is the output signal. Halting requires one computation step.

Method 4 requires the reservation of $k > 0$ states/nodes that are halting states/nodes. Theses states/nodes must be defined such that they are 1-cycles (all transitions from state/node q lead to q), and so cannot do any meaningful computation. Only one computation step is required to detect the halt.

Method 5 requires the reservation of $k > 0$ states/nodes that are halting states/nodes. Theses states/nodes must be defined such that they are 1-cycles (all transitions from state/node q lead to q), and so cannot do any meaningful computation. The automaton is assumed to have halted if:

- output is identical for the last d steps, and
- the state/node has remained constant for the last d steps.

The problem is that if the automaton is coupled to a Turing platform, such a sequence of transitions may be indistinguishable from a sequence of state transitions in a non-halting state. In fact, detecting a halted automaton in this manner is a problem equivalent to the halting problem for Turing machines. d computation steps are needed to determine whether the state is assumed to be a halting state or not.

DEFINITION 5 (DEFINED HALT) *An automaton has a* defined halt *if any host executing the automaton can explicitly detect the end of a computation by comparing the state to a predefined value, or comparing the output to a predefined value (methods 3 and 4 above).*

DEFINITION 6 (PROPERLY DEFINED AUTOMATON) *An automaton with a finite control is* properly defined *if it has a defined halt, and its state-transition and output mappings are defined for all possible combinations of defined states and defined inputs.*

3.3 Augmenting and Obfuscating Automata

There are four reasons why one would want to augment or obfuscate automata prior to encryption:

1. the automaton is to be encrypted in a tabular representation, and the tabular from of its initial state transition/output function has undefined entries;

2. one wishes to introduce spurious transitions/output and/or symbols as a private randomizer prior to encryption;

3. the automaton is to be used interactively such that an explicit halting state is required; and

4. some entries in the tabular representation are incomplete, making both polynomial and tabular representations impossible.

3.3.1 Augmenting Mealy Machines

Section 3.2 highlighted a problem with the classical definition of Mealy machines—there is no way of explicitly detecting a halt. This does not have to be a problem if explicit halting signals are employed. Otherwise, augmentation is a necessity if a host platform is to detect a halt.

This augmentation may be accomplished by adding a new state that is the halting state of the Mealy machine. To this end, one symbol must be selected as the "blank" symbol.

Denote by D the domain of the state transition and output mappings of a fixed Mealy machine M. The augmentation is performed as follows:

1. If M does not have an output symbol B reserved as a "blank" symbol, add a new symbol B (which cannot equal any symbol in Σ) to the output alphabet , setting $\Delta' = \Delta \cup \{B\}$, otherwise set $\Delta' = \Delta$, and call the previously reserved "blank" symbol B.

2. If M has a state $q \in Q$ such that for all inputs σ, $(q, \sigma) \notin D$, then call the state q_a and define $Q' = Q$. If M has a state $q \in Q$ such that for all inputs σ, $\delta(q, \sigma) = q$, call the state q_a, set $\lambda(q_a, \sigma) = \sigma$ for all inputs , and define $Q' = Q$. Otherwise:

   (a) add a new state q_a, such that $Q' = q_a \cup Q$,

   (b) for every node $q \neq q_a$ such that $\delta(q, \sigma) = q$ for all inputs (q, σ), set $\delta(q, \sigma) = q_a$ and $\lambda(q, \sigma) = \sigma$ for every $\sigma \in \Sigma$.

3. For every pair $(q, \sigma) \in D$ such that either $\delta(q, \sigma)$ or $\lambda(q, \sigma)$ or both are incompletely defined, set $\delta(q, \sigma) = q_a$, and $\lambda(q, \sigma) = B$.

Q' is the set of states of M'. The state q_a is henceforth referred to as the *augmentation state*. M' is the augmented Mealy machine.

EXAMPLE 1 *As an example of augmentation, consider a Mealy machine defined by the finite control of a universal Turing machine in [30] with four states and six tape symbols. The initial definition (with minor cosmetic changes) taken from [30] is*

	State			
Input	*0*	*1*	*2*	*3*
0	$(0, 4, -1)$	$(1, 1, -1)$	$(0, 5, 1)$	$(1, 5, -1)$
1	$(0, 4, -1)$	$(1, 0, 1)$	$(2, 1, 1)$	$(3, 0, 1)$
2	$(0, 3, 1)$	$(2, 3, -1,)$	$(3, 4, 1)$	$(1, 5, -1)$
3	$(0, 2, -1)$	$(1, 4, 1)$	$(2, 2, 1)$	$(3, 4, 1)$
4	$(0, 0, 1)$	$(1, 3, -1)$	$(2, -, -)$	$(3, -, -)$
5	$(3, 0, 1)$	$(1, 2, 1)$	$(0, 1, 1)$	$(3, 2, 1)$

*where each table entry is a (*next-state*,* output*,* move*) triple. All entries are partially defined in this automaton. It is not possible to use it directly, however, due to the transitions that are only partially defined. This alone is reason to augment the definition. Also, as there is no explicit signalling defined to notify the host-platform of a halt.*

Select as blank output symbol "0,0". Applying the augmentation described above results in one new state, and a new definition as follows:

	State				
Input	*0*	*1*	*2*	*3*	*4*
0	$(0, 4, -1)$	$(1, 1, -1)$	$(0, 5, 1)$	$(1, 5, -1)$	$(4, 0, 0)$
1	$(0, 4, -1)$	$(1, 0, 1)$	$(2, 1, 1)$	$(3, 0, 1)$	$(4, 0, 0)$
2	$(0, 3, 1)$	$(2, 3, -1,)$	$(3, 4, 1)$	$(1, 5, -1)$	$(4, 0, 0)$
3	$(0, 2, -1)$	$(1, 4, 1)$	$(2, 2, 1)$	$(3, 4, 1)$	$(4, 0, 0)$
4	$(0, 0, 1)$	$(1, 3, -1)$	$(4, 0, 0)$	$(4, 0, 0)$	$(4, 0, 0)$
5	$(3, 0, 1)$	$(1, 2, 1)$	$(0, 1, 1)$	$(3, 2, 1)$	$(4, 0, 0)$.

3.3.2 Obfuscation as a Private Randomizer

It is possible to obfuscate each of the automata types prior to any encryption. The obfuscations described in this dissertation depend on the N selected for the encryptable representation being such that there exists:

- one or more unused input symbols;
- one or more unused output symbols; and/or
- the possibility of adding states/nodes to the automaton.

Some methods of obfuscation, which do not depend on such additions will also be introduced, although they should not have cryptographical significance without the additions.

Common for all three automata types is that their encryptable representations work with inputs in $\mathbb{Z}_N^I$, state data in $\mathbb{Z}_N^S$ (with register automata having additional internal storage), and produce outputs in $\mathbb{Z}_N^O$. The representation of inputs, outputs, and state as vectors over $\mathbb{Z}_N$ is hereafter referred to as the *vectorized representation*.

The appropriate selection of N can depend on M itself and the desired scope for obfuscation. If M's encryptable representation is to be tabular, then it might be even be necessary to introduce additional states, input symbols, and/or output symbols merely to complete the function table[1].

Fix N and an automaton M. If M is a Mealy machine, assume that it already has been augmented as described in section 3.3.1.

One of the possible obfuscation techniques involves the addition of dummy states, dummy input symbols, and/or dummy output symbols. There are at least four ways of doing this:

1. No use is made of any initially redundant states in the definition of M in its vectorized representation. Undefined entries may be marked as such, but are left alone. This may only be done if M is represented using polynomials.

2. Dummy states are added until there are N^S defined states. Dummy input symbols are added until are N^I defined symbols. Dummy output symbols are added until there are N^O symbols. The exact way to do this varies with the automaton type:

   - M is a Mealy machine: For each pair $(q, \sigma) \notin D$, set $\delta(q, \sigma) = q_a$ and $\lambda(q, \sigma) = B$, where B is the fixed "blank" symbol in the output alphabet Δ.

[1] This is not a problem if M is a BSS' automaton, as it must necessarily have a polynomial representation. Converting the polynomial representation to a complete tabular representation is trivial.

- M is a BSS′ automaton: As long as $\mathcal{N} < N$, add another node. Define the next-node mapping such that the node is an isolated 1-cycle. Define the computation mapping g_n for the node so that it outputs the representation of a "blank" output, irrespective of input, and does not change the state vector.
- M is a register automaton: if M has initially been given a tabular definition, then it is possible to complete that definition in a manner similar to that for Mealy machines, otherwise the nature of the register automaton does not give much scope for dummy additions of this sort. There is however, the possibility, of adding dummy storage cells, and dummy program instructions in the program.

3. Randomized dummy states, inputs, and outputs are added. The following is done until there are no undefined states left:

   - M is a Mealy machine:
     (a) Select an existing state q at random. Add a new state q'.
     (b) For every input $\sigma \in \Sigma$, set $\delta(q', \sigma) = \delta(q, \sigma)$ and $\lambda(q', \sigma) = \lambda(q, \sigma)$.
     (c) Optionally: For every pair $(q, \sigma) \in D$ such that $\delta(q, \sigma) = q$, randomly set $\delta(q', \sigma)$ to q or q' and randomly set $\delta(q, \sigma)$ to q or q'.
   - M is a BSS′ automaton:
     (a) Select an existing node n at random. Add a new node n'.
     (b) Define the computation mapping $g_{n'}$ at node n' to be the identical to the one at node n.
     (c) Optionally: For every state vector–input combination at node n resulting in the next node being n, randomly define the next node as being node n or n'.
   - M is a register automaton: If the register automaton has an incomplete tabular definition, it can be completed in a manner similar to that of Mealy machines. There is not much scope for

obfuscation with the polynomial form. Both forms can employ randomly defined dummy storage cells and program instructions.

Add dummy input symbols until all possible input representations are defined (not applicable to register automata given on polynomial form or BSS' automata). Add dummy output symbols until all possible input representations are defined (not applicable to register machines given on polynomial form or BSS' automata). Next do the following:

- M is a Mealy machine: For each pair (q, σ), which so far has undefined next state and output, set $\delta(q, \sigma)$ to a random state, and $\lambda(q, \sigma)$ to a random output symbol.
- M is a register automaton on tabular form: A similar completion as per Mealy machines can be done for each of the mappings used by the register automaton. Also, it is possible to add randomly defined storage cells and program instructions.

In addition to the above obfuscation, it is possible to permute states, inputs, and/or outputs. This is feasible only for automata defined by function tables, thus it applies primarily to (tabularly defined) Mealy machines, and to some degree to register automata. For Mealy machines denote by δ' and λ' the post-permutation state-transition and output mappings, respectively. For register machines denote by f', g', h', and q' the post-permutation counterparts of f, g, h, and q, respectively. The permutations may be made swap-by-swap or may be pre-computed.

1. Permutation of states:

   - M is a Mealy machine: When swapping a state q with another state q', set $\delta'(q, \sigma) = \delta(q', \sigma)$, and $\delta'(q', \sigma) = \delta(q, \sigma)$.
   - M is a register automaton: For this type of obfuscation, the state can be considered as defined by the vectors $\mathcal{R} = (\vec{R}_1, \ldots, \vec{R}_m)$. When swapping a register state $\mathcal{R}$ with another $\mathcal{R}'$, set $f'(\mathcal{R}, \ldots) =$

$f(\mathcal{R}', \dots)$ and $f'(\mathcal{R}', \dots) = f(\mathcal{R}, \dots)$, where $f(\mathcal{R}, \dots)$ denotes all function table definitions where the argument "begins" with $\mathcal{R}$. Analogous assignments are done for g, h, and q. Note, however, that if some of the registers are reserved as inputs, they must not be affected by these permutations.

2. Permutation of the input alphabet:

   - M is a Mealy machine: When swapping an input symbol σ with another input symbol σ', set $\delta'(q, \sigma) = \delta(q, \sigma')$ for every $q \in Q$ and vice-versa. Similarly, set $\lambda'(q, \sigma) = \lambda(q, \sigma')$ for every $q \in Q$ and vice-versa. The swaps may be made one by one, or may be entirely precomputed. This type of swapping, however, must have corresponding swaps in the output alphabet for any symbols used to represent state information. Changes must be made known to the host that is to execute any partially encrypted Mealy machine if they affect inputs to be made by the host platform. Also, the encrypting party should know how to use the obfuscated and encrypted program, necessitating the recording of the permutations employed.
   - M is a register automaton: For this type of obfuscation, the input is defined as one or more of the following:
     - one or more storage cells in $\{\vec{S}_{\vec{\iota}}\}$,
     - one or more program instructions in $\mathcal{P}$, and/or
     - one or more registers in $\mathcal{R}$.

     Permutation of inputs in registers is done in the same way as for state stored in registers. When swapping a program instruction $\vec{\mathcal{P}}_a$ with another instruction $\vec{\mathcal{P}}_b$, set $f'(\mathcal{R}, \vec{\mathcal{P}}_a, \vec{S}, \vec{C}, \vec{D}) = f(\mathcal{R}, \vec{\mathcal{P}}_b, \vec{S}, \vec{C}, \vec{D})$, and $f'(\mathcal{R}, \vec{\mathcal{P}}_b, \vec{S}, \vec{C}, \vec{D}) = f(\mathcal{R}, \vec{\mathcal{P}}_a, \vec{S}, \vec{C}, \vec{D})$. Analogous changes are made in the mappings g, h, and q as well. In addition, all $\vec{\mathcal{P}}_{\vec{\iota}} \in \mathcal{P}$ equal to $\vec{\mathcal{P}}_a$ must be set to $\vec{\mathcal{P}}_b$. When swapping an instruction located at $\vec{C}_a$ with another instruction located at $\vec{C}_b$, set $f'(\mathcal{R}, \vec{\mathcal{P}}_{\vec{C}_a}, \vec{S}, \vec{C}_a, \vec{D}) = f(\mathcal{R}, \vec{\mathcal{P}}_{\vec{C}_b}, \vec{S}, \vec{C}_b, \vec{D})$, and

$f'(\mathcal{R}, \vec{\mathcal{P}}_{\vec{C}_b}, \vec{S}, \vec{C}_b, \vec{D}) = f(\mathcal{R}, \vec{\mathcal{P}}_{\vec{C}_a}, \vec{S}, \vec{C}_a, \vec{D})$. When swapping a storage cell located at $\vec{D}_a$ with another located at $\vec{D}_b$, both functioning as inputs, set $f'(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}_{\vec{D}_a}, \vec{C}, \vec{D}_a) = f(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}_{\vec{D}_b}, \vec{C}, \vec{D}_b)$, and $f'(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}_{\vec{D}_b}, \vec{C}, \vec{D}_b) = f(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}_{\vec{D}_a}, \vec{C}, \vec{D}_a)$. Analogous changes, where applicable, are made in the mappings g, h, and q as well.

3. Permutation of the output alphabet:

   - M is a Mealy machine: When swapping an output symbol x with another output symbol x', set $\lambda'(q, \sigma) = x'$ for every pair (q, σ) such that $\lambda(q, \sigma) = x$, and vice versa. Similar restrictions apply to this operation as with the permutation of the input alphabet. Changes must be made known to the host that is to execute the partially encrypted Mealy machine if they affect outputs to the remote host platform. Also, the encrypting party should know how to use the obfuscated and encrypted program, necessitating the recording the permutations employed.
   - M is a register automaton: For this type of obfuscation, the output is defined as one or more of the following:
     - one or more storage cells in $\{\vec{S}_{\vec{\iota}}\}$, and/or
     - one or more registers in $\mathcal{R}$.

     Since output via the registers effectively gives allows the host platform to read some of the machine's state directly, permuting these outputs is handled by the permutation of the state data stored in the registers. When swapping an output symbol $\vec{S}_a$ with another output symbol $\vec{S}_b$, for all tuples $(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}, \vec{C}, \vec{D}_a)$ such that $q(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}, \vec{C}, \vec{D}_a) = \vec{S}_a$, set $q'(\mathcal{R}, \vec{\mathcal{P}}, \vec{S}, \vec{C}, \vec{D}_a) = \vec{S}_b$, and vice-versa.

3.4 Interactive Mealy Machines

There are two principally different levels of interactivity:

1. passive, where only host storage is available to the Mealy machine in the form of a Turing platform or similar construction; and

2. active, where the Mealy machine sends some or all of its output to an oracle, and receives some or all of it from the same oracle, and possibly also has a Turing platform at its disposal.

For Mealy machines, the input is simply the input vector $\vec{y}(i)$, where the input alphabet is given a representation over $\mathbb{Z}_N$, such that $|\mathbb{Z}_N^I| \geq |\Sigma|$. Similarly, the output alphabet is given a representation over $\mathbb{Z}_N$ such that $|\mathbb{Z}_N^O| \geq |\Delta|$. The state is given a representation over $\mathbb{Z}_N$ such that $|\mathbb{Z}_N^S| \geq |Q|$.

For function table representations, it is a requirement that $|\Sigma| = |\mathbb{Z}_N^I|$, $|\Delta| = |\mathbb{Z}_N^O|$, and $|Q| = |\mathbb{Z}_N^S|$ prior to any encryption with the methods detailed in chapter 4.

When this is done, interactivity is defined by:

1. designating one or more components of the output vector as inputs to some oracle $\mathbb{O}$, and

2. designating one or more components of the input vector as receiving output from $\mathbb{O}$.

Next, it is necessary to define under which conditions a given Turing platform may interface with a given Mealy machine. This is done using the concept of a *compatible platform*.

DEFINITION 7 (COMPATIBLE PLATFORM (FOR MEALY MACHINES)) *A Turing platform $T = (\Gamma, C, P, D, W)$ is* compatible *with a Mealy machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ if there exists a mapping:*

1. *from M's output alphabet Δ to Γ,*

2. *from Γ to a subset of M's input alphabet Σ, and*

3. *from M's output alphabet, with a domain of at least three elements, to $\{-1, 0, 1\}$.*

THEOREM 1 *For every Turing machine t, there exists a pair (M, T) such that M is a Mealy machine, T a compatible Turing platform, M reads all its input from T's tape, and (M, T) simulates t.*

PROOF: Fix a Turing machine $t = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Fix a Mealy machine $M = (Q, \Sigma, \Gamma, \delta', \lambda', q_0)$, and a compatible Turing platform $T = (\Gamma, C, D, W)$. Define the domain of δ' as the union of the domain of δ and $F \times \Sigma$. For every pair (q, σ) in the domain of δ:

1. set $\delta'(q, \sigma) = \delta_1(q, \sigma)$; and
2. set $\lambda(q, \sigma) = (\delta_2(q, \sigma), h(\delta_3(q, \sigma)))$, where $h(\text{“right”}) = 1$ and $h(\text{“left”}) = -1$.

For every pair (q, σ) not in the domain of δ, but in $F \times \Sigma$:

1. set $\delta'(q, \sigma) = q$; and
2. set $\lambda(q, \sigma) = (\sigma, 0)$.

Since the Mealy machine does not have a defined set of final states, the halting action of the Turing machine is simulated by no further movement of the Turing platform, and no further changes on the tape. □.

COROLLARY 2 *For every pair (M, T) such that M is a Mealy machine, T a compatible Turing platform, and M reads all its input from T's tape, there exists a Turing machine t which simulates (M, T).*

PROOF: (M, T) defines what is in this dissertation referred to as a Turing-like machine (see appendix B.1). By theorem 55, there exists a turing machine t which simulates the defined Turing-like machine. □.

COROLLARY 3 *There exists at least one pair (M, T), such that M is a Mealy machine, T is a compatible Turing platform, and (M, T) simulates a universal Turing machine.*

PROOF: by construction. Use the small universal Turing machine constructed in [30] with four states and six tape symbols. Augment this machine as shown in example 1.

By theorem 1, a pair (M, T) simulating this Turing machine exists. □.

3.5 Interactive BSS′ Automata

Originally, BSS′ automata do non-interactive computations on an input in order to produce an output. Their computations are non-interactive in the sense that only an initial input is given, and no further input is accepted during subsequent applications of the computing endomorphism. Some other finite state automata may be regarded as interactive in the sense that they accept input with each application of the state transition function. It might be desirable to introduce this type of "interactivity" into BSS′ automata over finite fields. This means that there is input for each application of the computing endomorphism.

In addition, the possibility of allowing output for each application of the computing endomorphism would be desirable. This may be achieved by doing the following:

- reserving I components of the state-space for input; as a convention components $x_{d-I+1}, \ldots, x_d$ are selected
- reserving O components of the state-space for output; as a convention components $x_{d-I-O+1}, \ldots, x_{d-I}$ are selected
- selecting component x_C for use in computing b_K, where C is such that $2 \leq C \leq d - I - O$.

The resulting revised full state space is thus:

$$\mathbb{S} = \mathbb{Z}_N \times \mathbb{Z}_N^S \times \mathbb{Z}_N^O \times \mathbb{Z}_N^I = \mathbb{Z}_N^{d+1}. \tag{3.9}$$

Denote by $g_n(\vec{x})_i$ the i^{th} component of $g_n(\vec{x})$. Incorporating the above modifications places restrictions on g_n. Since $x_{d-I+1}, \ldots, x_d$ now are inputs, define $g_n(\cdot)_i = x_i$ for $d - I + 1 \leq i \leq d$, so that inputs are not modified by computation. In addition, require

$$\frac{\partial g_n(\cdot)_i}{\partial x_j} \equiv 0$$

for $i \leq d - I - O$ and $d - I - O + 1 \leq j \leq d - I$. This ensures that no computations depend on data already output. Because of this, the computing endomorphism for BSS' automata will henceforth be considered a mapping

$$H : \mathbb{Z}_N^{1+S+I} \longrightarrow \mathbb{Z}_N^{S+O}.$$

To make the use of input and output consistent, it is required that:

1. All nodes accept input from the last I components in the full state vector $\vec{x} \in \mathbb{S}$.

2. All nodes compute output to components number $1 + S + 1$ through $1 + S + O$.

DEFINITION 8 (COMPATIBLE PLATFORM (FOR BSS' AUTOMATA)) *A Turing platform $T = ()$ is* compatible *with a BSS' automaton M if:*

1. *there exists a bijection from a subspace M_Γ of M's output alphabet $\mathbb{Z}_N^O$ to Γ,*

2. *there exists a mapping from Γ to a subspace of M's input alphabet $\mathbb{Z}_N^I$, and*

3. *there exists a subspace M_D of M's output alphabet $\mathbb{Z}_N^O$ which contains at least three elements, and a surjection which maps it to $\{-1, 0, 1\}$, and $M_D \cap M_\Gamma = \vec{0}$.*

THEOREM 4 *For every Turing machine t, there exists a pair (M, T), such that M is a BSS' automaton, T a compatible Turing platform, M reads all its input from T's tape, and (M, T) simulates t.*

PROOF: Fix a Turing machine $t = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Let M be a BSS' automaton with $\mathcal{N} = |Q|$ nodes, and select N, S, and O such that: $N \geq \mathcal{N}$, $S = 0$ (the node number functions as the state), $N \geq |\Sigma|$, and $N^O \geq |\Gamma|$. Construct a mapping $f : Q \longrightarrow \mathbb{Z}_\mathcal{N}$, a mapping $g : \Sigma \longrightarrow \mathbb{Z}_N$, and a mapping $h : \Gamma \times \{L, R\} \longrightarrow \mathbb{Z}_N^O \times \{-1, 1\}$. Select $C = S + 2$, so that the number in the input is used as the second argument of the next node function. For

every pair (q, σ) in the domain of δ set $\beta(f(q), g(\sigma)) = f(\delta_1(q, \sigma))$. For each node $f(q)$, such that $q \in Q$, interpolate $g_{f(q)}$ such that for all pairs (q, σ) in the domain of δ, $g_{f(q)}(g(\sigma)) = h(\delta_2(q, \sigma), \delta_3(q, \sigma))$. In addition, for every pair $(q, \sigma) \in F \times \Sigma$, $g_{f(q)}$ must satisfy $g_{f(q)}(g(\sigma)) = (g(\sigma), 0)$. As with the Mealy machine, the halting is simulated by an infinite loop that does not change the position of the head of the Turing platform's finite control, and that does not change the contents of the tape after a state in F has been entered $\square$.

COROLLARY 5 *For every pair (M, T), such that M is a BSS' automaton, T a compatible Turing platform, and M reads all its input from T's tape, there exists a Turing machine t which simulates (M, T).*

PROOF: M can be assumed to have a polynomial definition, and thus has an associated defined power of a prime N. Construct a table for a finite control with $\mathcal{N} \cdot N^S$ (S being the number of state vector components of M excluding the node number) columns and N^I rows. Apply the computing endomorphism to compute entries of the form (*new state,output,move*) such that output is an alphabet with N^{O-1} symbols. The remaining output component of M defines the move. The resulting finite control defines a Turing-like machine, which by theorem 55 can be simulated by a standard Turing machine $\square$.

COROLLARY 6 *There exists at least one pair (M, T), such that M is a BSS' automaton, T is a compatible Turing platform, and (M, T) simulates a universal Turing machine.*

PROOF: by construction. Use the small universal Turing machine constructed in [30] with four states and six tape symbols. Although, strictly speaking, not necessary, it is helpful to augment the definition prior to use, as a BSS' automaton requires some defined halting state. Therefore use the augmented definition produced in example 1.

By theorem 4, a pair (M, T) simulating this Turing machine exists $\square$.

3.6 Interactive Register Automata

Register automata can be made interactive in a number of ways, by making registers and/or storage cells available to external entities. For coupling to a Turing platform the following additions are natural:

1. The specification of a register dedicated to output of movement direction, in the form of an integer y such that $0 < y \leq m$, and an integer z such that $0 < z \leq d$. The integer y indicates the register, and the integer z, the component in which this movement is stored.

2. The specification of a register dedicated as output to the storage cells of a Turing platform's tape.

3. The specification of a register dedicated as input from a Turing platform.

The definition of the register automaton given in section already incorporates the elements necessary to facilitate communication with an oracle $\mathbb{O}$. The computation itself only specifies explicitly which cells are used for input from the host—including both the oracle and any Turing platform at the automaton's disposal. The oracle may read any storage cell or any register at will, entirely asynchronously to M's execution, as long as reads are between individual step in the computation. Whether or not the oracle can interpret the data it reads, is determined by any encryption template used to encrypt M. The register automaton allows an additional type of encryption not possible with Mealy or BSS' automata.

DEFINITION 9 (COMPATIBLE PLATFORM (FOR REGISTER AUTOMATA)) *A Turing platform $T = (\Gamma, C, D, W)$ is* compatible *with a Register automaton M if:*

1. *there exists a mapping from $\mathbb{Z}_N^d$ to Γ,*

2. *there exists a mapping from Γ to $\mathbb{Z}_N^d$,*

3. *there exists a surjection from $\mathbb{Z}_N^d$ to $\{-1, 0, 1\}$, and*

4. *M has at least three registers.*

THEOREM 7 *For every Turing machine t, there exists a pair (M, T), such that M is a register automaton, T a compatible Turing platform, M reads all its input from T's tape, and (M, T) simulates t.*

PROOF: Fix a Turing machine $t = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Let M be a register automaton such that $N^d \geq |\Gamma|$, $N^{(m-3)d} \geq |Q|$, Construct a mapping: $a : Q \longrightarrow \mathbb{Z}_N^{ds}$, where $\mathbb{Z}_N^{d(s-1)} < |Q| \leq \mathbb{Z}_N^{ds}$, a mapping $b : \Sigma \longrightarrow \mathbb{Z}_N^d$, a mapping $c : \Gamma \times \{L, R\} \longrightarrow \mathbb{Z}_N^d \times \{-1, 1\}$. The first s registers will store the state, the next register the output, the register after that the direction, and lastly the input. Any remaining registers are unused. Define a single instruction vector $\vec{\mathcal{P}}_{\vec{0}}$. Set $\vec{C} = \vec{0}$. Define the next instruction pointer mapping f to be the constant $\vec{0} \in \mathbb{Z}_N^d$. For every pair (q, σ) in the domain of δ set

$$(h_1(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}), \ldots, \\ h_{ds}(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D})) = a(\delta_1(q, \sigma)).$$

For every pair (q, σ) in the domain of δ set

$$(h_{ds+1}(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}), \ldots, \\ h_{d(s+2)}(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D})) = c(\delta_2(q, \sigma), \delta_3(q, \sigma)).$$

For every pair $(q, \sigma) \in F \times \Sigma$ set

$$(h_{ds+1}(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}), \ldots, \\ h_{d(s+2)}(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D})) = (b(\sigma), 0),$$

and set

$$(h_1(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D}), \ldots, \\ h_{ds}(a(q), \ldots, b(\sigma), \vec{\mathcal{P}}_{\vec{C}}, \vec{S}_{\vec{D}}, \vec{C}, \vec{D})) = a(q).$$

The halting is simulated by an infinite loop, that does not change the tape contents of the Turing platform, or change the position of the finite control of the Turing platform. □.

COROLLARY 8 *For every pair (M, T), such that M is a register automaton, T a compatible Turing platform, and M reads all its input from T's tape and does not use its storage cells or program instructions, there exists a turing machine which simulates (M, T).*

PROOF: Fix (M, T). M must have $m - k$ registers reserved for input, and $s + 1$ registers used for output and indicating movement. The remaining registers record state. Construct a table with $N^{d(k-s-1)}$ columns, and $N^{d(m-k)}$ rows. Write into each entry a triple (*next-state*,*output*,*move*). The resulting table defines finite control of a Turing-like machine. By theorem 55, there exists a standard Turing machine which simulates this machine $\square$.

COROLLARY 9 *There exists at least one pair (M, T), such that M is a register automaton, T is a compatible Turing platform, and (M, T) simulates a universal Turing machine.*

PROOF: by construction. Use the small universal Turing machine constructed in [30] with four states and six tape symbols. To fix the problem with the undefined states and lack of defined method of halting, augment the definition as in example 1.

By theorem 7, a pair (M, T) simulating this Turing machine exists $\square$.

Chapter 4

Encrypting Functions Using Composition

This chapter describes three related cryptosystems based on symbolic function composition. They appear to have fairly similar strengths and weaknesses.

4.1 Privacy Homomorphisms Revisited

Privacy homomorphisms were originally conceived as a method of processing encrypted data (see [29] and [17]). In more recent times, it has been proposed as a principle underlying encrypted computation [32].

The mapping doing M's computation was given in equation 3.3. Denote this mapping by H. Since $\vec{y}(n)$ is given by Bob, H is such that

$$H : \mathbb{Z}_N^{S+I} \longrightarrow \mathbb{Z}_N^{S+O}.$$

If Alice is to keep her machine, and thus H, secret, along with any state information, H and $\vec{x}$ must somehow be encrypted, and not appear in plaintext at Bob's host. Privacy homomorphisms were originally proposed as a solution to this problem, based on an encryption function E_K and a

decryption function D_K. The basic principle for the scenario in this paper would be to give Bob:

1. the encrypted data $a', b', \ldots$ of the plaintext data $a, b, \ldots$, where $a = D_K(a') = D_K(E_K(a))$,
2. an encrypted version f' of each function f, defined as
$$f'(a', b', \ldots) = E_K(f(D_K(a'), D_K(b'), \ldots)), \tag{4.1}$$
3. an encrypted version p' of each predicate p, defined as
$$p'(a', b', \ldots) \text{ iff } p(D_K(a'), D_K(b'), \ldots).$$

Bob would then use this to do Alice' computation.

The encryption functions E_K and D_K are then *privacy homomorphisms* when such is achieved. Unfortunately, they result in inherently weak cryptographic systems (see [17], page 159). This is due to the fact that Bob is given an ordering for the encrypted data, and is in addition given the *separate* encrypted versions of each operation to be applied to the data.

Instead of giving Bob the separate encrypted versions of the operations and predicates, it is only necessary to give Bob *one* encrypted mapping, which does Alice' *entire* computation. A good candidate is a computing endomorphism like H. Given such a mapping, Bob does not need to know of any orderings of the encrypted data, which makes it infeasible to do searches like the one presented in [17].

Essentially, it boils down to the problem of encrypting mappings using symbolic function composition. It is unlikely that all function classes are suited to encryption through composition. The function classes that so far have been found to suit such encryption are polynomials over finite fields, and function tables.

4.2 Univariate Encryption

Fix a number N and a positive integer m. Fix a multivariate mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$, which is the mapping to be encrypted. If N is a power of

a prime, f may be represented either as a polynomial in $\mathbb{Z}_N[x_1, \ldots, x_m]$ or as a function table indexed by vectors in $\mathbb{Z}_N^m$, having values in $\mathbb{Z}_N^n$. If N is not a power of a prime, f is assumed to be represented as a function table.

To encrypt f, select a permutation $r : \mathbb{Z}_N \longrightarrow \mathbb{Z}_N$. The permutation r is assumed to have the same representation as f. The inverse of r, s, is also represented in the same way as r. There are N pairs $(a, b) \in \mathbb{Z}_N \times \mathbb{Z}_N$ defining r. Thus a polynomial of degree $N - 1$ is sufficient for exact interpolation of all permutation values, when N is a power of a prime.

In a departure from data encryption, there is no one fixed expression for a mapping encrypted using symbolic functional composition. As a matter of fact, the above requirements for privacy homomorphisms are unnecessarily restrictive, as it is possible to operate with only partially encrypted data. An example will illustrate this.

EXAMPLE 2 *Let h be bivariate, so $h \in \mathbb{Z}_N[x_1, x_2]$. Denote by g the polynomial constructed by encrypting f using symbolic function composition. Note that $y_1 = r(x_1)$ and $y_2 = r(x_2)$, that is: y_1 and y_2 are encrypted data. Encryption by composition may then take the following forms:*

- $g(y_1, y_2) = r(h(s(y_1), s(y_2)))$
- $g(y_1, y_2) = h(s(y_1), s(y_2))$
- $g(y_1, y_2) = r(h(s(y_1), s(y_2)))$
- $g(x_1, y_2) = r(h(x_1, s(y_2)))$
- $g(y_1, x_2) = r(h(s(y_1), x_2))$
- $g(x_1, x_2) = r(h(x_1, x_2))$
- $g(x_1, y_2) = h(x_1, s(y_2))$
- $g(y_1, x_2) = h(s(y_1), x_2)$

In general, one wishes to encrypt multivariate mappings. For bivariate polynomial mappings, each component will either be unencrypted or on one of the forms given in example 2. More generally, one wishes to encrypt

f in a similar fashion. At most $n + m$ different key pairs can be applied when encrypting f to produce

$$E(f) = (r_1(f_1(s_{n+1}(x_1), \ldots, s_{n+m}(x_m))), \ldots, \\ r_n(f_n(s_{n+1}(x_1), \ldots, s_{n+m}(x_m)))), \quad (4.2)$$

where $E(f)$ is a simplistic and imprecise way of referring to the encrypted result. Henceforth, a plaintext function f will be called the *functional plaintext*. Its encrypted counterpart, $E(f)$, will be called the *functional ciphertext*. Plaintext data input to or output by $E(f)$ are called *plaintext*, and ciphertext data input to or output by $E(f)$ are called *ciphertext*. Before encrypting f, one needs to

- select the subset of variables to be decrypted (as they are assumed to be *encrypted* when $E(f)$ is applied), by selecting a subset $I \subseteq \{n + 1, \ldots, n + m\}$ of key indexes corresponding to the variable indexes; and
- select the subset of mapping components to be encrypted, by selecting a subset $J \subseteq \{1, \ldots, n\}$ of key indexes corresponding to the component indexes.

The key pairs with indices not in $I \cup J$ are defined to be the identity mapping. The remaining keys with indices in $I \cup J$ may or may not be restricted in some manner. For the purposes of encrypted computation, some typical restrictions are equality restrictions, requiring certain pairs of keys to be equal. A consequence is that the (possibly partially) encrypted mapping may be tailored to:

- take specified inputs in ciphertext and others in plaintext, and
- produce specified outputs in ciphertext and others in plaintext.

DEFINITION 10 (UNIVARIATE ENCRYPTION TEMPLATE) *A tuple* (m, n, N, I, J) *where:*

- m *is the arity (number of variables) of the mappings it applies to,*

- *n is the number of components of the mappings it applies to,*
- *N is the order of the ring over which the mappings it applies to are defined,*
- *$I \subseteq \{n+1, \ldots, n+m\}$ contains the indices of variables to be used in encrypted form plus n, and*
- *$J \subseteq \{1, \ldots, n\}$ contains the indices of the mapping components to be encrypted.*

The next step is generating the expressions for the non-identity permutations $r_i : \mathbb{Z}_N \longrightarrow \mathbb{Z}_N$ and computing their inverses s_i.

The encryption itself is done by symbolically substituting each x_i with $s_{n+i}(x_i)$, effectively composing each f_j with $s_{n+1}, \ldots, s_{n+m}$. In addition, each f_j is composed with r_j. The resulting expression in 4.2 is a mapping with the same number of components and variables as f originally had. It also has the same representation, domain, and range as f.

Decryption is done by symbolically substituting each x_i in $E(f)$ with $r_{n+i}(x_i)$, effectively composing each f_j with $s_{n+1}, \ldots, s_{n+m}$. In addition, each f_j is composed with s_j. The resulting expression is

$$\begin{aligned}(s_1(r_1(f_1(s_{n+1}(r_{n+1}(x_1)), \ldots, s_{n+m}(r_{n+m}(x_m))))), \ldots, \\ s_n(r_n(f_n(s_{n+1}(r_{n+1}(x_1)), \ldots, s_{n+m}(r_{n+m}(x_m)))))) \\ = (f_1(\vec{x}), \ldots, f_n(\vec{x}))\end{aligned} \quad (4.3)$$

4.3 Univariate Key Regeneration

It is possible to remotely re-encrypt any functional ciphertext. There is the following result for the cryptosystem presented in the previous section:

THEOREM 10 *Fix a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$, a univariate encryption template U such that no key pair is set to the identity, and two sets of key pairs $K = \{(r_i, s_i)\}_{i=1}^{n+m}$, and $K' = \{(r'_i, s'_i)\}_{i=1}^{n+m}$. Initially, only party A knows U,*

K, and K'. Key pairs are assumed to be uniformly distributed. Denote by $E(f)$ the encryption of f with the keys in K, and by $E(f)'$ the encryption of f with the keys in K'. Then the following hold:

1. *$E(f)$ can be transformed by a party B into $E(f)'$.*
2. *If B does not know I or J, B gains no knowledge about f from $E(f)'$ and the process of generating it, that could not have been gained from $E(f)$.*

PROOF: Note that U fixes the set I of indexes of the variables that are decrypted, and the set J of indexes of function components that are encrypted. This ensures that previously undecrypted variables and unencrypted function components remain thus if A generates both sets of key pairs correctly.

CLAIM 1: Party A generates the following functions using symbolic composition:

$$G = \{r'_1 \circ s_1, \ldots, r'_n \circ s_n, r_{n+1} \circ s'_{n+1}, \ldots, r_{n+m} \circ s'_{n+m}\}.$$

A then sends to B: m, n, N, and G. B substitutes variable i, x_i, in $E(f)$ with $r_{n+i}(s'_{n+m}(x_i))$. Next, B composes $r'_j \circ s_j$ with the jth function component of $E(f)$. This gives the mapping:

$$\begin{aligned}
&(r'_1(s_1(r_1(f_1(s_{n+1}(r_{n+1}(s'_{n+1}(x_1))), \ldots, s_{n+m}(r_{n+m}(s'_{n+m}(x_m)))))))), \ldots,\\
&\quad r'_n(s_n(r_n(f_n(s_{n+1}(r_{n+1}(s'_{n+1}(x_1)))), \ldots, s_{n+m}(r_{n+m}(s'_{n+m}(x_m)))))))\\
&= (r'_1(f_1(s'_{n+1}(x_1), \ldots, s'_{n+m}(x_m))), \ldots, r'_n(f_n(s'_{n+1}(x_1), \ldots, s'_{n+m}(x_m))))\\
&= E(f)'. \quad (4.4)
\end{aligned}$$

Thus B is capable of generating $E(f)'$ from $E(f)$.

CLAIM 2: A secret key secrecy system is *perfect* if for all ciphertexts $E(f)$, the a posteriori probability that the ciphertext is generated from a plaintext f, after being read by the cryptanalyst, is equal to the a priori probability of $E(f)$ being generated from f (see [34]). The a priori probability that a composition $r'_i(s_i())$ is generated from (r_i, s_i) (viewing (r'_i, s'_i)

as the encryption key) is $(N!)^{-1}$. The a posteriori probability that $r'_i(s_i())$ is generated from (r_i, s_i) is the inverse of the number of possible decompositions of $r'_i \circ s_i$ into two permutations a and b such that $a \circ b = r'_i \circ s_i$. There are $N!$ such possible decompositions. To see this, start by selecting a from all $N!$ permutations. Then $b = a^{-1} \circ r'_i \circ s_i$ is a solution which always exists, as the permutation group containing a must also contain its inverse. Thus a priori and a posteriori probabilities remain equal, and Shannon's requirement for perfect secrecy is satisfied. Since all key pairs are randomly selected using a uniform distribution, the list G does not reveal any information about keys in K or K'. Therefore B learns nothing about f that could not have been learned from $E(f)$ _____ □.

4.4 Cryptanalysis of Univariate Encryption

Encryption with univariate functions is characterized by the following:

1. It is an asymmetric *secret* key algorithm—neither of the keys in a key-pair (r, s) may be publicized, as knowing one allows the construction of the other.

2. For every set of key pairs and encryption generated with the key pairs, there are *two types* of plaintext and ciphertext:

   (a) The mapping f represents one of the types of plaintext, and $E(f)$ its encrypted equivalent.

   (b) The datum $\vec{x}$ represents the other type of plaintext, and $(E(f))(\vec{x})$ its *transformed* equivalent. Note that $(E(f))(\vec{x})$ is not necessarily encrypted, a fact which is not necessarily known by the attacker.

LEMMA 11 *Fix a univariate encryption template U. If the cryptanalyst has the function table representation for r_i or s_i, but not both, for the i^{th} key pair, the cryptanalyst can find the tabular representation for s_i or r_i, respectively, in time $\mathcal{O}(N)$.*

PROOF: Without loss of generality, assume the cryptanalyst knows r_i. If r_i is represented as a function table, it can be considered a list of pairs $(x, r_i(x))$, where $0 \leq x < N$. An ordered table for s_i can be generated in time linear in N using a simple radix sort. For every $(x, r_i(x))$ set $y = r_i(x)$ and $s_i(y) = x$ □.

COROLLARY 12 *Fix a univariate encryption template U. If the cryptanalyst has N distinct (*ciphertext,plaintext*) pairs for the same encryption keys, the cryptanalyst can construct a table for both the relevant encryption and decryption keys, r_i and s_i, in time $\mathcal{O}(N)$.*

PROOF: r_i and s_i permute $\mathbb{Z}_N$. The tabular representations of r_i and s_i can be constructed using radix sort as described in the proof of lemma 11 □.

Univariate encryption admits attacks from more points than data encryption ciphers. As before, f is called the *functional plaintext*, $E(f)$ the *functional ciphertext*, x the plaintext, and its encrypted equivalent the ciphertext. $(E(f))(x)$ is called the *transformed plaintext/ciphertext* if $(E(f))(x)$ does not/does encrypt x after transforming it.

There are four new variants of attack directed at the functional ciphertext:

- *functional ciphertext-only attack*: the cryptanalyst knows only the encrypted function $E(f)$ and the public randomizer if it exists
- *functional known-plaintext attack*: the cryptanalyst knows some pairs $(f, E(f))$ for the current key(s)
- *functional chosen-plaintext attack*: the cryptanalyst can get functional ciphertexts for some chosen functional plaintexts
- *functional chosen-ciphertext attack*: the cryptanalyst can get functional plaintexts for some chosen function ciphertexts

This cryptanalysis proceeds by first applying approaches from the traditional four types of cryptanalytic attack, before applying approaches from the four types above. For the following sections fix a mapping f,

index sets I and J, and keys r_i and s_i for $i \in I \cup J$. Let $E(f)$ be fixed by these parameters on the form given in equation 4.2. The key pairs (r_i, s_i) may all be the same key pair or may be different.

THEOREM 13 *Univariate encryption of a functional plaintext is perfectly secure only for univariate encryption templates when $N > 2$, $m = 1$, $n = 1$, both key pairs are randomly and independently selected, and the functional ciphertext is applied only once.*

PROOF: By theorem 59, page 137, there are more plaintexts than possible keys when $N = 2$ and $n, m \geq 1$, and at least one of n and m is greater than one. Therefore perfect secrecy is impossible for these cases. When $N = 2$, $n = 1$, and $m = 1$, there are $(N^n)^{(N^m)} = 2^2 = 4$ different possible plaintexts. There are $(N!)^{n+m} = 2^2 = 4$ possible keys. The a priori probability that a particular $f : \mathbb{Z}_2 \longrightarrow \mathbb{Z}_2$ has been encrypted is $1/4$. The two possible permutations over $\mathbb{Z}_2$ are $x + 0$ and $x + 1$. These preserve the degree of f. Thus an attacker can, using the degree of $E(f)$, narrow the possibilities down to two, making the a posteriori probability $1/2$ for both the possible degrees of f. Therefore univariant encryption is not perfectly secure for this case either. By lemma 60, $(N^n)^{(N^m)} < (N!)^{n+m}$ whenever $N > 2$, and $m = n = 1$. Thus only in these cases is it possible to achieve perfect security for the functional plaintext $\square$.

4.4.1 Chosen-ciphertext Attack

In their conventional form, chosen-ciphertext attacks require the cryptanalyst to be capable of using the key(s) directly. This is also the case for univariate encryption.

The cryptanalyst can use the key directly without knowing its value[1]. Therefore the cryptanalyst can systematically generate pairs $(y_j, s_i(y_j))$, which can be used to interpolate s_i and r_i, or generate tables for s_i and r_i.

[1] An example is the attacker who knows the PIN to a smartcard, and thus can use the keys stored in it without necessarily being able to read the values of those keys.

THEOREM 14 *Fix a univariate encryption template U. If a cryptanalyst can apply, but not read either r_i or s_i for a selected subset $A \subseteq K$ of all key pairs $K = \{(r_i, s_i)\}_{i=1}^{n+m}$ applied to encrypt f using univariate encryption to give $E(f)$, then:*

1. *all the keys in A are vulnerable,*
2. *any ciphertext encrypted by a key in A, input to- or output by $E(f)$ is vulnerable, and*
3. *any components of $E(f)$ encrypted using keys in A are vulnerable.*

PROOF: Without loss of generality, the cryptanalyst begins with r_i.

CLAIM 1: The cryptanalyst can now generate pairs $(x_j, r_i(x_j))$ (with s_i, the cryptanalyst generates pairs $(s_i(y_j), y_j)$), which can be used to interpolate r_i or simply generate a function table for r_i as described in the proof of lemma 11. This can be done for every key belonging to a key pair in A.

CLAIM 2: Follows trivially from the proof of claim 1.

CLAIM 3: The cryptanalyst can recover f or parts of f from $E(f)$ by symbolically substituting x_i with $r_{n+i}(x_i)$ for all $1 \leq i \leq m$ and symbolically composing s_i with $E(f)_i()$ for all $1 \leq i \leq n$. The result is

$$\begin{aligned}(s_1(r_1(f_1(s_{n+1}(r_{n+1}(x_1)), \ldots, s_{n+m}(r_{n+m}(x_m))))), \ldots, \\ s_n(r_n(f_n(s_{n+1}(r_{n+1}(x_1)), \ldots, s_{n+m}(r_{n+m}(x_m)))))) \\ = (f_1(\vec{x}), \ldots, f_n(\vec{x})), \quad (4.5)\end{aligned}$$

which is the plaintext expression for f. Alternately, the cryptanalyst can recover *part* of f, if A is a proper subset of K ——————————— □.

COROLLARY 15 *Univariate encryption is vulnerable to chosen-ciphertext attacks.*

PROOF: Any chosen-ciphertext attack must allow the cryptanalyst use of the key pairs. By theorem 14, any ability to apply all the keys renders at least part of $E(f)$, generated through univariate encryption, vulnerable □.

4.4.2 Chosen-plaintext Attack

The results for this type of attack are very similar to those presented in subsection 4.4.1. The same set of cases and subcases in subsection 4.4.1 apply directly to the chosen plaintext case, and the analysis is for all practical purposes identical.

COROLLARY 16 *Univariate encryption is vulnerable to chosen-plaintext attacks.*

PROOF: Any chosen-plaintext attack must allow the cryptanalyst use of the key pairs. By theorem 14, any ability to apply all the keys renders $E(f)$, generated through univariate encryption, vulnerable$\square$.

4.4.3 Ciphertext-only Attack

The encryption permutations r_i viewed with respect to data encryption, are monoalphabetic substitution ciphers based on an N-letter alphabet. Given sufficient amounts of ciphertext and the language of the plaintext data, cryptanalytic attack based on frequency analysis becomes possible, provided the plaintext is non-uniformly distributed.

The number of ciphertext bits necessary to attempt effective cryptanalysis can be approximated using the expression for unicity distance by Shannon [34]:

$$n_u \approx \left\lceil \frac{H(r)}{\rho} \right\rceil . \tag{4.6}$$

The number of ciphertext blocks necessary to attempt effective ciphertext-only cryptanalysis for *one* unique key pair is thus:

$$b_u = \left\lceil \frac{H(r)}{\rho \log_2 N} \right\rceil \tag{4.7}$$

Assuming the random selection has a uniform distribution, the entropy of the key space (for one key pair) is given by

$$H(r) = N! \cdot \frac{1}{N!} \cdot -\log_2 \left(\frac{1}{N!} \right) = \log_2(N!) \tag{4.8}$$

The key length in bits as a function of N is approximately log:

$$l_r(N) = \lceil \log_2 N \rceil N, \tag{4.9}$$

which is more than the entropy computed in equation 4.8. Thus there is some redundancy in the key itself.

It is possible, to some extent, to avoid frequency-based cryptanalysis by regularly re-encrypting f using the method outlined in section 4.3. The re-encryption intervals are decided by computing the maximum number of times a key can be used before it becomes vulnerable to frequency-based attacks b_u. One then only has to divide b_u by the number of times the same key information is used in any one application of $E(f)$ to see how many times $E(f)$ can be applied before its keys must be changed. Note that in order for a scheme such as this to work fully, remotely stored data encrypted by $E(f)$ must be:

1. re-encrypted, which raises the number of times the new key is applied, unless it is possible to assume that such a use is carried out within a trusted computing base, and the new data are not given to cryptanalysts/attackers;

2. thrown away, to reduce the number of times the new key is applied; or

3. securely archived along with the old key.

LEMMA 17 *If a functional ciphertext $E(f)$, encrypted with univariate encryption, is applied more than b_u times without re-encryption, all keys are vulnerable to ciphertext-only attack.*

PROOF: b_u is the number of ciphertext blocks needed to mount a ciphertext-only attack $\square$.

Thus univariate encryption is vulnerable to this form of encryption whenever:

- there is a significant amount of redundancy in the plaintext language of the data, and/or

- $E(f)$ must be applied a large number of times to do its job.

4.4.4 Known-plaintext Attack

Instead of being able to apply the keys, the cryptanalyst merely has access to a limited number of (*ciphertext*,*plaintext*) pairs. If the cryptanalyst possesses N (*ciphertext*,*plaintext*) pairs for a given key pair, that key pair must be considered compromised (see corollary 12).

If the cryptanalyst possesses $k < N$ (*ciphertext*,*plaintext*) pairs for a given key pair, the cryptanalyst knows k of the N unknowns needed to find the keys. This can be exploited to attack functional ciphertext. If nothing else, the unicity distance is reduced, so that the number of ciphertext blocks needed to attempt subsequent ciphertext-only cryptanalysis effectively is:

$$b'_u = \left\lceil \frac{H(r')}{\rho \log_2(N-k)} \right\rceil, \tag{4.10}$$

where

$$H(r') = \log_2((N-k)!). \tag{4.11}$$

4.4.5 Functional Chosen-ciphertext Attack

The functional ciphertext is on the form in equation 4.2. Each encryption key may be written

$$r_j(x) = \sum_{i=0}^{N-1} A_{j,i} x^i. \tag{4.12}$$

Decryption keys may be written

$$s_j(x) = \sum_{i=0}^{N-1} B_{j,i} x^i. \tag{4.13}$$

Similarly, the plaintext function may in general be written

$$f_j(x_1, \ldots, x_m) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_m=0}^{N-1} \phi_{j,i_1,\ldots,i_m} x_1^{i_1} \ldots x_m^{i_m}. \tag{4.14}$$

The functional ciphertext $E(f)$ may be written

$$E(f)_j = \sum_{i_1=0}^{N-1} \cdots \sum_{i_m=0}^{N-1} \Phi_{j,i_1,\ldots,i_m} x_1^{i_1} \ldots x_m^{i_m}. \tag{4.15}$$

If N is not a power of a prime, $E(f)$ may be represented by two different polynomial mappings:

1. the composition product F given in equation 4.2, which has degree no more than $(N-1)^3$ in any given variable, no less than 1 in any given variable, and probably at least $(N-1)^2$ for any given variable;

2. the polynomial mapping G generated by interpolating the tabular representation of $E(f)$, which has degree no more than $N-1$ in any given variable.

Obviously, $F(\vec{x}) = G(\vec{x})$ for all $\vec{x} \in \mathbb{Z}_N^m$. Thus if N is a power of a prime, $E(F)$ is represented by the composition of the keys and f reduced modulo N (see appendix B.2), and otherwise, $E(f)$ is represented by G.

An efficient functional chosen-ciphertext attack is possible. This attack can also be used when the key pairs have a polynomial representation, provided the generated ciphertext functions are converted to polynomials prior to decryption.

THEOREM 18 *Fix a univariate encryption template U and key pairs $K = \{(r_i, s_i)\}_{i=1}^{n+m}$. If a cryptanalyst can generate plaintexts f from ciphertexts $E(f)$, that have been generated using U and K, then a functional chosen-ciphertext attack can be completed in asymptotic time $\mathcal{O}(NnN^m)$, excluding any time spent doing polynomial interpolation or converting polynomials to function tables, which is nearly linear in the number of coefficients nN^m needed to represent f using an interpolation polynomial.*

PROOF: In this case, the keys and functions are represented using function tables. To reveal the keys, the constant functions $0, \ldots, N-1$ are successively encrypted using U and K. Fix a constant $C \in \{0, \ldots, N-1\}$. For all

j, $1 \leq j \leq n$, $E(f)_j(\vec{x}) = C$. Thus

$$E(f)_j(r_{n+1}(x_1), \ldots, r_{n+m}(x_m)) = C,$$

and

$$f_j = s_j(E(f)_j(r_{n+1}(x_1), \ldots, r_{n+m}(x_m)) = s_j(C).$$

In this way it becomes possible to reconstruct in parallell all the permutations $s_1, \ldots, s_n$ (and thus also $r_1, \ldots, r_n$) using radix sorting of the type described in the proof of lemma 11.

The next step is to generate one (or more, depending on the actual arity of each component) $E(f)_j$ such that:

$$\begin{aligned}
E(f)_j(0, \ldots, 0) &= 0 \\
E(f)_j(1, 0, \ldots, 0) &= 1 \\
&\vdots \\
E(f)_j(N-1, 0, \ldots, 0) &= N-1 \\
E(f)_j(0, 1, 0, \ldots, 0) &= 1 \\
&\vdots \\
E(f)_j(0, N-1, 0, \ldots, 0) &= N-1 \\
&\vdots \\
E(f)_j(0, \ldots, 0, 1) &= 1 \\
&\vdots \\
E(f)_j(0, \ldots, 0, N-1) &= N-1,
\end{aligned} \tag{4.16}$$

and $E(f)_j(\vec{x}) = 0$ otherwise. Note that all the arguments in equation 4.16 are encrypted. When the resulting $E(f)$ is decrypted to functional plain-

text, f_j will be defined as follows:

$$\begin{aligned}
f_j(s_{n+1}(1), s_{n+2}(0), \ldots, s_{n+m}(0)) &= s_j(1) \\
&\vdots \\
f_j(s_{n+1}(N-1), s_{n+2}(0), \ldots, s_{n+m}(0)) &= s_j(N-1) \\
f_j(s_{n+1}(0), s_{n+2}(1), s_{n+3}(0), \ldots, s_{n+m}(0)) &= s_j(1) \\
&\vdots \\
f_j(s_{n+1}(0), s_{n+2}(N-1), s_{n+3}(0), \ldots, s_{n+m}(0)) &= s_j(N-1) \\
f_j(s_{n+1}(0), \ldots, s_{n+m-1}(0), s_{n+m}(1)) &= s_j(1) \\
&\vdots \\
f_j(s_{n+1}(0), \ldots, s_{n+m-1}(0), s_{n+m}(N-1)) &= s_j(N-1)
\end{aligned} \tag{4.17}$$

Since $s_1, \ldots, s_n$ are known, the right-hand side can be re-encrypted, so as to construct pairs $(x_i, s_j(x_i))$ when $n < j \leq n + m$. Using radix sort again, sorted tables for the remaining keys can be constructed.

Functions encryptable under U are defined by tables with N^{n+m} entries. Generating the tables for the first half of the attack takes $\mathcal{O}(nN^{m+1})$ time. Decrypting them also takes $\mathcal{O}(nN^{m+1})$ time. Generating the table for the second half of the attack takes $\mathcal{O}(nN^m)$ time. Decrypting it takes $\mathcal{O}(nN^m)$ time. Asymptotic time complexity is therefore roughly $\mathcal{O}(nN^{m+1})$, which is N times the number of coefficients needed to represent a interpolation polynomial over $\mathbb{Q}$ for any f encryptable under U □.

COROLLARY 19 *Univariate encryption is vulnerable to functional chosen-ciphertext attack.*

PROOF: Follows from theorem 18 □.

4.4.6 Functional Chosen-plaintext Attack

The analysis is similar to that in section 4.4.5, and so is the result.

COROLLARY 20 *Univariate encryption is vulnerable to functional chosen-plaintext attack.*

PROOF: Follows from theorem 18 □.

4.4.7 Functional Ciphertext-only Attack

There are two obvious ways to attack a functional ciphertext:

1. direct decomposition of each component of $E(f)$, and/or
2. accumulation of enough functional ciphertexts $E(f), E(g), \ldots$ to do a frequency-based attack to find the key.

The second alternative is practical if:

- the key pairs, along with the univariate encryption template, do not vary with each encryption, so that accumulation of sufficient amounts of functional ciphertext encrypted with the same sets of key pairs is feasible;
- the plaintext language has a fair amount of redundancy, and
- the cryptanalyst is capable of recognizing the plaintext.

This analysis will concentrate on decomposition attacks. Univariate encryption is based on the assumption that decomposing $E(f)$ to find one or more of the functions $f_1, \ldots, f_n$, $r_1, \ldots, r_{n+m}$, $s_1, \ldots, s_{n+m}$ is infeasible.

In its simplest form, the problem is symbolically decomposing the univariate function $E(f) = r \circ f$ to find f and r. As in section 4.4.5, $E(f)$ is represented by the reduced composition of f and the relevant keys if N is a power of a prime, and by G otherwise. Henceforth denote by F the composition $r \circ f$.

In general f is an element of the semigroup $K = (\mathbb{Z}_N[x], \circ)$, where $\circ$ is the symbolic functional composition operation. In general r is an element in S_N, the group of permutations of $\mathbb{Z}_N^1$. S_N is a sub-semigroup of K. All elements in S_N have inverses, but elements in $K - \mathbb{P}$ do not necessarily have inverses. Furthermore, K is not in general commutative for $N > 2$. Also, composition is non-linear, so $f \circ (ag)$ is not in general equal to $a(f \circ g)$,

when a is a scalar. Decomposing $E(f)$ thus amounts to searching K and S_N for an f and r, respectively, such that $r \circ f = E(f)$.

By lemma 57, $r \circ f$ can always be reduced to degree $\leq N-1$, even though $\deg r \cdot \deg f \geq N-1$. Thus only when $\deg r \cdot \deg f \leq N-1$ does one have a case of the so-called Univariate Decomposition Problem (here taken from [18] with cosmetic modifications).

PROBLEM 7 (UNIVARIATE DECOMPOSITION) *Given a monic polynomial $f(x) \in K[x]$ of degree n, K a commutative ring with identity, and integers a and b such that $n = ab$ and $a, b > 1$, decide if there exists a functional decomposition g, h of f (such that $f = g \circ h$) with $\deg g = a$ and $\deg h = b$. If so, determine the coefficients of g and h.*

LEMMA 21 *Given $E(f) = r \circ f$ such that $\deg E(f) \leq N-1$, and $\deg r \cdot \deg f \leq N-1$, then $E(f)$ is vulnerable to ciphertext-only attacks.*

PROOF: Since $\deg r \cdot \deg f \leq N-1$, decomposing $E(f)$ is an instance of the Univariate Decomposition (problem 7). Kozen and Landau present in [23] an algorithm that solves this problem, and decomposes $E(f)$ in time $\mathcal{O}((\deg r \cdot \deg f)^2 (\deg r))$ $\square$.

When $\deg r \cdot \deg f \geq N$, then one has another problem:

DEFINITION 11 (REDUCED POLYNOMIAL) *A polynomial $f \in \hat{K}[x_1, \ldots, x_m]$, where $\hat{K}$ is an algebraic extension of a commutative ring K, is called* reduced *if $\deg f < |K|$ for every one of f's variables, and there exists another polynomial $\hat{f} \in K[x_1, \ldots, x_m]$ such that $f(\vec{x}) = \hat{f}(\vec{x})$, for all $\vec{x} \in K^m$, and $\deg \hat{f} > \deg f$. $\hat{f}$ is then* reducible *to f.*

For finite fields, it is always possible to find a reduction of $\hat{f}$ in $K[x_1, \ldots, x_m]$, so in those cases, $\hat{K} = K$.

With reduced polynomials, instead of problem 7, one faces the following problem:

PROBLEM 8 (REDUCED UNIVARIATE DECOMPOSITION) *Given a polynomial $f(x) \in \hat{K}[x]$ of degree n and integers r and s such that $n < |K| \in \mathbb{N}$ and*

$n \leq rs$ and $r, s > 1$, decide if there exist two polynomials $g, h \in K[x]$ such that $\deg g = r$, $\deg h = s$, and $g(h(x))$ is reducible to f. If so, determine the coefficients of g and h.

Next, consider another case, where $E(f) = (f \circ (s_1, \ldots, s_m)$, such that all $\deg f \cdot \deg s_i \leq N - 1$ when $1 \leq i \leq m$. This is an instance of Simple Multivariate Decomposition (again taken from [18] with cosmetic modifications).

PROBLEM 9 (SIMPLE MULTIVARIATE DECOMPOSITION) *Given a monic multivariate polynomial $f(\vec{x}) \in K[\vec{x}]$ with $\deg_i f = n$ for every variable number i, integers $a > 1$ and b such that $ab = n$, decide if there exists a functional decomposition g, h of f with g univariate, $\deg g = a$, and $\deg_i h = b$ for every variable number i. If so, determine the coefficients of g and h.*

LEMMA 22 *Given $E(f) = f \circ (s_1, \ldots, s_m)$ such that $\deg E(f) \leq N - 1$, $\deg f = a$ in all variables, and $\deg s_i \cdot \deg f \leq N - 1$ when $1 \leq i \leq m$, then $E(f)$ is vulnerable to ciphertext-only attacks.*

PROOF: Since $\deg s_i \cdot \deg f \leq N - 1$ when $1 \leq i \leq m$, and $\deg f = a$ in all variables, decomposing $E(f)$ is an instance of Simple Multivariate Decomposition (problem 9). Dickerson presents in chapter 2 in [18] an algorithm with asymptotic time complexity $\mathcal{O}(N^{3m})$ for decomposing $E(f)$, where N^m is the number of coefficients in f's representation. For fields $\mathbb{Z}_N$ supporting fast Fourier transforms, Dickerson presents an algorithm with asymptotic time complexity $\mathcal{O}((\deg s_i)^m N^m \log_2(\deg f))$ ———— □.

Again, if $\deg f \cdot \max\{\deg s_i\} > \deg E(f)$ in any variable, then one is not dealing with a case of problem 9 anymore. Indeed, the algorithms that solve problems 7 and 9 use the leading coefficients of the unreduced composition. If a reduction has occurred, then for either of the special cases above: $r \circ f$ and $f \circ (s_1, \ldots, s_m)$, there are at least $\deg r \cdot \deg f - N$ and $(\deg f) \sum_{i=1}^{m} (\deg s_i) - N^m$ coefficients "missing", respectively. Therefore one also has the following problem:

PROBLEM 10 (REDUCED SIMPLE MULTIVARIATE DECOMPOSITION) *Given a multivariate polynomial $f(\vec{x}) \in \hat{K}[\vec{x}]$, $\vec{x} \in K^m$, having degree no greater than n in any variable and degree equal to n in at least one variable (and total degree $\leq mn$), and given integers r and s such that $n \leq rs$ and $r > 1$, decide if there exist polynomial mappings $g \in K[y]$, and $h \in K[\vec{x}]$ such that $\deg g = r$, h has degree no greater than s in any of its variables, and degree equal to s for at least one variable, and such that $g(h(\vec{x}))$ is reducible to f. If so, determine the coefficients of g and h.*

The following relation holds for all j such that $1 \leq j \leq n$ and all vectors $(i_1, \ldots, i_m) \in \mathbb{Z}_N^m$:

$$\Phi_{j,i_1,\ldots,i_m} \equiv \psi_{j,\vec{t}}(A_{j,0}, \ldots, A_{j,N-1}, B_{n+1,0}, \ldots, B_{n+1,N-1}, \ldots, B_{n+m,0}, \ldots, B_{n+m,N-1}, \phi_{j,0,\ldots,0}, \ldots, \phi_{j,N-1,\ldots,N-1}) (\mod N). \quad (4.18)$$

Each $\psi_{j,\vec{t}}$ is the general expression for the reduced composition of r_j with f_j, which in turn may have one or more of its variables substituted by the corresponding s_k. There are thus nN^m equations with $nN^m + (n+m)N$ independent unknowns. The $(n+m)N$ unknowns are the encryption keys. Since

$$\deg r_j \cdot \deg f_j \cdot \deg s_i \geq N,$$

many of the equivalences of the form in equation 4.18 will be non-linear in many variables. The non-linearity implies that solution techniques such as Gauss elimination do not work. Furthermore, since there may be as many as $(n+m)N$ degrees of freedom in the solution, it is not necessarily "simply" a question of solving the system of equations. A brute force search must check $N^{(nN^m)}N!(n+m)$ possible solutions for all nN^m equations.

LEMMA 23 *Given a functional composition $E(f) : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ with coefficients $\Phi_{j,\vec{t}}$ satisfying equation 4.18, with at least one encryption key having degree $N \geq 5$ for at least one of its variables, there does not exist any analytic method of solving the system of equations resulting from the relation in equation 4.18.*

PROOF: The univariate encryption of a function has the form:

$$\sum A_{j,i} \left(\sum \phi_{j,\vec{\iota}} (s_{n+1}(x_1))^{i_1} \cdots (s_{n+m}(x_m))^{i_m} \right)^i \tag{4.19}$$

for each component j. For $\vec{\iota} = \vec{0}$, $(s_{n+k}(x_k))^{i_k} = 1$, since all $i_k = 0$, $1 \leq k \leq m$. Thus at least n variables, $\phi_{j,\vec{0}}$, will exist in at least 6 possible powers: $0, \ldots, 5$. Thus there is at least one equation for each of these n variables of degree 5. By Abel's well-known result on polynomial equations, there are no general analytic solutions for polynomial equations having degree 5 or more in at least one variable. Thus there does not exist any general analytic method of arriving at a solution for the system of equations $\square$.

The consequence of lemma 23 is that attacks on $E(f)$ that make use of the system of equations can only attempt direct solution by approximate methods (or brute force-type methods).

4.4.8 Functional Known-plaintext Attack

The cryptanalyst has access to a limited number of (*functional ciphertext*, *functional plaintext*) pairs.

LEMMA 24 *Fix a univariate encryption template U and a set of key pairs $K = \{(r_i, s_i)\}_{i=1}^{n+m}$. If the cryptanalyst knows one (*functional ciphertext,functional plaintext*) pair for U and K, then any other functional ciphertext $E(f)$ generated using U and K is compromised.*

PROOF: The knowledge of the single pair gives knowledge of the coefficients $\Phi_{j,\vec{\iota}}$ and $\phi_{j,\vec{\iota}}$ for all $j \in \{1, \ldots, n\}$ and $\vec{\iota} \in \mathbb{Z}_N^m$. Setting up the relations given by equation 4.18 results in a system of up to nN^m equations with $(n+m)N$ independent unknowns, and $2(n+m)N$ dependent unknowns. Thus at most $2(n+m)N$ equations need solving to reveal all the keys in K $\square$.

Thus univariate encryption is vulnerable to functional known-plaintext attack.

4.5 Multivariate Encryption

Fix an integer $N > 1$, positive integers m and n, a positive integer $k \leq n + m$, and a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. Select a sequence of integers $\{c_i\}_{i=1}^k$ such that $\sum_{i=1}^k c_i = n + m$ and $\sum_{i=1}^l c_i = n$ for some $l < k$. The selected series of integers c_i represents "block sizes" used in encryption and decryption. Thus one may encrypt directly a mapping:

$$f(x_1, \ldots, x_m) = (f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m)) \tag{4.20}$$

over the integers modulo N. If N is a power of a prime, f, along with any encryption and decryption keys, may be represented either using function tables or polynomials over $\mathbb{Z}_N$. Otherwise, f is assumed to be represented using a function table. The variables of f are grouped into blocks of $c_{l+1}, c_{l+2}, \ldots, c_k$ variables as follows:

$$\underbrace{x_1, \ldots, x_{c_{l+1}}}_{\text{block } l+1}, \underbrace{x_{c_{l+1}+1}, \ldots, x_{c_{l+1}+c_{l+2}}}_{\text{block } l+2}, \ldots, \underbrace{x_{m-c_k+1}, \ldots, x_m}_{\text{block } k}. \tag{4.21}$$

For notational convenience, write the i^{th} variable block as $\vec{w}_i$.

The components of f are grouped into blocks of c_1-, c_2-,. . . , c_l-component mappings as follows:

$$(\underbrace{f_1, \ldots, f_{c_1}}_{\text{block 1}}, \underbrace{f_{c_1+1}, \ldots, f_{c_1+c_2}}_{\text{block 2}}, \ldots, \underbrace{f_{n-c_l+1}, \ldots, f_n}_{\text{block } l}). \tag{4.22}$$

Some examples will hopefully make this easier to grasp.

EXAMPLE 3 *Assume $f \in \mathbb{Z}_N[x_1, x_2, x_3, x_4]^5$. The mapping f can also be written $(f_1, f_2, f_3, f_4, f_5)$. One possible grouping of function components is:*

$$(\underbrace{f_1, f_2, f_3}_{\textit{block 1}}, \underbrace{f_4, f_5}_{\textit{block 2}}). \tag{4.23}$$

Another possible grouping of function components is:

$$(\underbrace{f_1, f_2}_{\textit{block 1}}, \underbrace{f_3, f_4}_{\textit{block 2}}, \underbrace{f_5}_{\textit{block 3}}). \tag{4.24}$$

More combinations are possible, and by selecting only one component per block, subsequent multivariate encryption of the function components is effectively reduced to univariate encryption. Similarly, one possible grouping of variables, assuming the first grouping of function components above is selected, is:

$$\underbrace{x_1, x_2}_{\text{block 3}}, \underbrace{x_3, x_4}_{\text{block 4}}. \tag{4.25}$$

If the second grouping of function components had been selected, the above grouping of variables would have looked like this:

$$\underbrace{x_1, x_2}_{\text{block 4}}, \underbrace{x_3, x_4}_{\text{block 5}}.$$

As with the function components, other groupings are possible, and ultimately, it is possible to reduce multivariate decryption of variables to univariate decryption of variables, by only assigning one variable to each variable block.

The general expression for a partially encrypted mapping f, encrypted using key triples generated according to a multivariate encryption template, is then:

$$\begin{aligned} E(f) = \quad & (r_1(f_1(s_{l+1}(\vec{w}_1), \ldots, s_k(\vec{w}_{k-l})), \ldots, \\ & f_{c_1}(s_{l+1}(\vec{w}_1), \ldots, s_k(\vec{w}_{k-l}))), \ldots, \\ & r_l(f_{n-c_l+1}(s_{l+1}(\vec{w}_1), \ldots, s_k(\vec{w}_{k-l})), \ldots, \\ & f_n(s_{l+1}(\vec{w}_1), \ldots, s_k(\vec{w}_{k-l})))) \end{aligned} \tag{4.26}$$

where $E(f)$ once again is a simplistic (and imprecise) way of writing the encryption.

EXAMPLE 4 *Continuing example 3, multivariate encryption of f, using the grouping given in equation 4.23 would in general be on the form*

$$\begin{aligned} E(f) = r_1(f_1(s_3(\vec{w}_1), s_4(\vec{w}_2)), f_2(s_3(\vec{w}_1), s_4(\vec{w}_2)), f_3(s_3(\vec{w}_1), s_4(\vec{w}_2))), \\ r_2(f_4(s_3(\vec{w}_1), s_4(\vec{w}_2)), f_5(s_3(\vec{w}_1), s_4(\vec{w}_2))). \end{aligned} \tag{4.27}$$

Assuming that s_4 and r_2 are the identity mappings, equation 4.27 may be written:

$$\begin{aligned} E(f) = r_1(f_1(s_3(\vec{w}_1), \vec{w}_2), f_2(s_3(\vec{w}_1), \vec{w}_2), f_3(s_3(\vec{w}_1), \vec{w}_2)), \\ f_4(s_3(\vec{w}_1), \vec{w}_2), f_5(s_3(\vec{w}_1), \vec{w}_2)). \end{aligned} \tag{4.28}$$

Before encrypting f, one needs to

- define the sizes of the variable blocks;
- define the sizes of the function component blocks;
- select the subset of variable blocks to be encrypted, by selecting a subset $I \subseteq \{l+1, \ldots, k\}$ of the variable block indices; and
- select the subset of mapping component blocks to be encrypted, by selecting a subset $J \subseteq \{1, \ldots, l\}$ of the component block indices.

Denote by F_j the j^{th} mapping component block $f_a, \ldots, f_{a+c_j-1}$, where $a = \sum_{i=1}^{j-1} c_i$. Since block sizes may vary, keys are defined by triples, and for the sake of simplicity, these triples will henceforth be referred to as "key triples". Key triples whose indices are not in $I \cup J$ are defined to be the identity mapping. The remaining key triples with indices in $I \cup J$ may or may not be restricted in some manner, as for univariate encryption. This allows the tailoring of the encrypted mapping to:

- take specified blocks of inputs in ciphertext and other blocks in plaintext, and
- produce specified blocks of outputs in ciphertext and other blocks in plaintext.

DEFINITION 12 (MULTIVARIATE ENCRYPTION TEMPLATE) *A tuple* $(m, n, N, I, J, k, \{c_i\}_{i=1}^{k})$ *where:*

- m *is the arity (number of variables) of the mappings it applies to,*
- n *is the number of components of the mappings it applies to,*
- N *is the order of the ring over which the mappings it applies to are defined,*
- $I \subseteq \{l+1, \ldots, k\}$ *contains the indices of blocks of variables to be used in encrypted form,* l *is such that* $\sum_{i=1}^{l} c_i = n$*,*

- $J \subseteq \{1, \ldots, l\}$ *contains the indices of the blocks of mapping components to be encrypted,*
- k *is the total number of blocks defined, and*
- $\{c_i\}_{i=1}^{k}$ *are the block sizes themselves.*

The next step is generating the non-identity permutations $r_i : \mathbb{Z}_N^{c_i} \longrightarrow \mathbb{Z}_N^{c_i}$, and computing their inverses. Thus $r_i = (r_{i,1}, \ldots, r_{i,c_i})$ and $s_i = (s_{i,1}, \ldots, s_{i,c_i})$.

The encryption is done by symbolically substituting each $\vec{w}_i$ with $s_{l+i}(\vec{w}_i)$, effectively composing each f_j with $s_{l+1}, \ldots, s_k$. In addition, each block of function components $(f_a, \ldots, f_{a+c_j-1})$, $a = \sum_{i=1}^{j-1} c_i$ is composed with r_j. The resulting expression in equation 4.26 is a mapping with the same number of components and variables as f originally had. It also has the same representation, domain, and range as f.

Decryption is done by symbolically substituting each $\vec{w}_i$ in $E(f)$ with $r_{l+i}(\vec{w}_i)$, effectively composing each f_j with $r_{l+1}, \ldots, r_k$. In addition, each block of function components $(F_a, \ldots, F_{a+c_j-1})$, $a = \sum_{i=1}^{j-1} c_i$ is composed with s_j. The resulting expression is:

$$\begin{aligned}
&(s_1(r_1(f_1(s_{l+1}(r_{l+1}(\vec{w}_1)), \ldots, s_k(r_k(\vec{w}_{k-l}))), \ldots, \\
&\qquad f_{c_1}(s_{l+1}(r_{l+1}(\vec{w}_1)), \ldots, s_k(r_k(\vec{w}_{k-l})))))), \ldots, \\
&\; s_l(r_l(f_{n-c_l+1}(s_{l+1}(r_{l+1}(\vec{w}_1)), \ldots, s_k(r_k(\vec{w}_{k-l}))), \ldots, \\
&\qquad f_n(s_{l+1}(r_{l+1}(\vec{w}_1)), \ldots, s_k(r_k(\vec{w}_{k-l})))))) \\
&\qquad = (f_1(\vec{x}), \ldots, f_n(\vec{x})) = f.
\end{aligned} \tag{4.29}$$

4.6 Multivariate Key Regeneration

It is possible to remotely re-encrypt any functional ciphertext. There is the following result for the cryptosystem presented in the previous section:

THEOREM 25 *Fix a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$, a multivariate encryption template U such that no key triple is set to the identity, and two sets of key triples $K = \{(c_i, r_i, s_i)\}_{i=1}^{n+m}$, and $K' = \{(c_i, r'_i, s'_i)\}_{i=1}^{n+m}$. Initially, only*

party A knows U, K, and K'. Key triples are assumed to be uniformly distributed. Denote by $E(f)$ the encryption of f with the keys in K, and by $E(f)'$ the encryption of f with the keys in K'. Then the following hold:

1. *$E(f)$ can be transformed by a party B into $E(f)'$.*
2. *If B does not know I or J, B gains no knowledge about f from $E(f)'$ and the process of generating it, that could not have been gained from $E(f)$.*

PROOF: Note that U fixes the set I of indices of the blocks of variables that are decrypted, and the set J of indices of the blocks of function components that are encrypted. This ensures that previously undecrypted variables and unencrypted function components remain thus if A generates both sets of key triples correctly.

CLAIM 1: Party A generates the following functions using symbolic composition:

$$G = \{r'_1 \circ s_1, \ldots, r'_n \circ s_n, r_{n+1} \circ s'_{n+1}, \ldots, r_{n+m} \circ s'_{n+m}\}.$$

A then sends to B: m, n, N, k, $\{c_i\}_{i=1}^{k}$, and G. B substitutes variable block number i, $\vec{w}_{i-l}$, in $E(f)$ with $r_i(s'_i(\vec{w}_{i-1}))$. Next, B composes $r'_j \circ s_j$ with the jth function component of $E(f)$. This gives the mapping:

$$\begin{aligned}
&(r'_1(s_1(r_1(f_1(s_{l+1}(r_{l+1}(s'_{l+1}(\vec{w}_1))), \ldots, s_k(r_k(s'_k(\vec{w}_{k-l})))), \ldots,\\
&\qquad f_{c_1}(s_{l+1}(r_{l+1}(s'_{l+1}(\vec{w}_1))), \ldots, s_k(r_k(s'_k(\vec{w}_{k-l}))))))), \ldots,\\
&\ r'_l(s_l(r_l(f_{n-c_l+1}(s_{l+1}(r_{l+1}(s'_{l+1}(\vec{w}_1), \ldots, s_k(r_k(s'_k(\vec{w}_{k-l})))), \ldots,\\
&\qquad f_n(s_{l+1}(r_{l+1}(s'_{l+1}(\vec{w}_1), \ldots, s_k(r_k(s'_k(\vec{w}_{k-l}))))))) \qquad (4.30)\\
&= (r'_1(f_1(s'_{l+1}(\vec{w}_1), \ldots, s'_k(\vec{w}_{k-l}))), \ldots,\\
&\quad r'_l(f_l(s'_{l+1}(\vec{w}_1), \ldots, s'_k(\vec{w}_{k-l})))) = E(f)'.
\end{aligned}$$

Thus B is capable of generating $E(f)'$ from $E(f)$.

CLAIM 2: A secret key secrecy system is *perfect* if for all ciphertexts $E(f)$, the a posteriori probability that the ciphertext is generated from a plaintext f, after being read by the cryptanalyst, is equal to the a priori

probability of $E(f)$ being generated from f (see [34]). The a priori probability that a composition $r'_i \circ s_i$ is generated from (r_i, s_i) (viewing (r'_i, s'_i) as the encryption key) is $((N^{c_i})!)^{-1}$. The a posteriori probability that $r'_i \circ s_i$ is generated from (r_i, s_i) is the inverse of the number of possible decompositions of $r'_i \circ s_i$ into two permutations a and b such that $a \circ b = r'_i \circ s_i$. There are $(N^{c_i})!$ such possible decompositions. To see this, start by selecting a from all $(N^{c_i})!$ permutations. Then $b = a^{-1} \circ r'_i \circ s_i$ is a solution which always exists, as the permutation group containing a must also contain its inverse. Thus a priori and a posteriori probabilities remain equal, and Shannon's requirement for perfect secrecy is satisfied. Since all key pairs are randomly selected using a uniform distribution, the list G does not reveal any information about keys in K or K'. Therefore B learns nothing about f that could not have been learned from $E(f)$ ——————— □.

4.7 Cryptanalysis of Multivariate Encryption

Multivariate encryption shares some of the characteristics of univariate encryption:

1. It is an asymmetric secret key algorithm—neither of the keys r_i or s_i in a key triple (c_i, r_i, s_i) may be publicized, as knowing one allows the construction of the other. Furthermore, keeping c_i secret may in some cases make cryptanalysis more difficult.

2. For every set of key triples and encryption generated with the key triples, there are *two types* of plaintext and ciphertext:

   (a) The mapping f represents one of the types of plaintext, and $E(f)$ its encrypted equivalent.

   (b) The datum $\vec{x}$ represents the other type of plaintext, and $(E(f))(\vec{x})$ its *transformed* equivalent.

LEMMA 26 *Fix a multivariate encryption template U. If the cryptanalyst has the function table representation for r_i or s_i, but not both, for the i^{th}*

key triple, the cryptanalyst can find the tabular representation for s_i or r_i, respectively, in time $\mathcal{O}(N^{c_i})$.

PROOF: Without loss of generality, assume the cryptanalyst knows r_i. If r_i is represented as a function table, it can be considered a list of pairs $(\vec{w}, r_i(\vec{w}))$, where $\vec{w} \in \mathbb{Z}_N^{c_i}$. An ordered table for s_i can be generated in time linear in N^{c_i} using a simple radix sort. For every $(\vec{w}, r_i(\vec{w}))$ set $\vec{v} = r_i(\vec{w})$ and $s_i(\vec{v}) = \vec{w}$ □.

COROLLARY 27 *Fix a multivariate encryption template U. If the cryptanalyst has N^{c_i} distinct (*ciphertext,plaintext*) pairs for the same encryption keys, the cryptanalyst can construct a table for both the relevant encryption and decryption keys, r_i and s_i, in time $\mathcal{O}(N^{c_i})$.*

PROOF: r_i and s_i permute $\mathbb{Z}_N^{c_i}$. The tabular representations of r_i and s_i can be constructed using radix sort as described in the proof of lemma 26 □.

LEMMA 28 *Multivariate encryption is perfectly secure when $N > 2$, $c_1 = n$, $c_2 = m$, ($k = 2$, $l = 1$) both key triples are randomly and independently selected, and the functional ciphertext is applied only once.*

PROOF: Follows from lemma 60 □.

There exist $(N^n)^{(N^m)}$ different mappings $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. At most $\prod_{i=1}^{k}(N^{c_i})!$ pairs of keys may be independently selected per encryption. For only one application, multivariate encryption has the possibility of being perfectly secure as long as $(N^n)^{(N^m)} \leq \prod_{i=1}^{k}(N^{c_i})!$. This is more general than the result in the lemma above, but is not a particularly useful result.

4.7.1 Chosen-ciphertext Attack

As with univariate encryption, chosen-ciphertext attacks require the cryptanalyst to be capable of using the key(s) directly.

As with univariate encryption, the cryptanalyst can use the key directly without knowing its value. In this case, the cryptanalyst can systematically generate pairs $(\vec{w}_i, s_i(\vec{w}_i))$, which can be used to interpolate s_i and

r_i. This attack is efficient enough, as N and the c_i's cannot be extremely large, lest the legitimate user be swamped by the system's complexity.

THEOREM 29 *Fix a multivariate encryption template U. If a cryptanalyst can apply, but not read either r_i or s_i for a selected subset $A \subseteq K$ of all key triples $K = \{(c_i, r_i, s_i)\}_{i=1}^{k}$ applied to encrypt f using multivariate encryption to give $E(f)$, then:*

1. *all the keys in A are vulnerable,*
2. *any ciphertext encrypted by a key in A, input to- or output by $E(f)$ is vulnerable, and*
3. *any components of $E(f)$ encrypted using keys in A are vulnerable.*

PROOF: Without loss of generality, the cryptanalyst begins with r_i.

CLAIM 1: The cryptanalyst can now generate pairs $(\vec{w}_j, r_i(\vec{w}_j))$ (with s_i, the cryptanalyst generates pairs $(s_i(\vec{w}_j), \vec{w}_j)$), which can be used to interpolate r_i or simply generate a function table for r_i. This can be done for every key belonging to a key triple in A.

CLAIM 2: Follows trivially from the proof of claim 1.

CLAIM 3: The cryptanalyst can recover f, or parts of f from $E(f)$ by symbolically substituting $\vec{w}_i$ with $r_{l+i}(\vec{w}_i)$ for all $1 \leq i \leq k-l$ and symbolically composing s_i with function component block i for all $1 \leq i \leq l$. The result is

$$\begin{aligned}(s_1(r_1(f_1(s_{l+1}(r_{l+1}(\vec{w}_1)), \ldots, s_k(r_k(\vec{w}_{k-l}))))), \ldots, \\ s_l(r_l(f_l(s_{l+1}(r_{l+1}(\vec{w}_1)), \ldots, s_k(r_k(\vec{w}_{k-l})))))) \\ = (f_1(\vec{x}), \ldots, f_n(\vec{x})), \quad (4.31)\end{aligned}$$

which is the plaintext expression for f. Alternately, the cryptanalyst can recover *part* of f, if A is a proper subset of K ——————————— □.

COROLLARY 30 *Multivariate encryption is vulnerable to chosen-ciphertext attacks.*

PROOF: Any chosen-ciphertext attack must allow the cryptanalyst use of the key pairs. By theorem 29, any ability to apply all the keys renders $E(f)$, generated through multivariate encryption, vulnerable_____ □.

4.7.2 Chosen-plaintext Attack

The results for this type of attack are very similar to those presented in subsection 4.7.1. The same set of cases and subcases in subsection 4.7.1 apply directly to the chosen plaintext case, and the analysis is for all practical purposes identical.

COROLLARY 31 *Multivariate encryption is vulnerable to chosen-plaintext attacks.*

PROOF: Any chosen-plaintext attack must allow the cryptanalyst use of the key pairs. By theorem 29, any ability to apply all the keys renders $E(f)$, generated through multivariate encryption, vulnerable_____ □.

4.7.3 Ciphertext-only Attack

The encryption permutations r_i viewed with respect to data encryption, are monoalphabetic substitution ciphers based on a N^{c_i}-letter alphabet. Given sufficient amounts of ciphertext and the language of the plaintext data, cryptanalytic attack based on frequency analysis becomes possible, provided plaintext is non-uniformily distributed.

The number of ciphertext bits necessary to attempt effective cryptanalysis of block number i can be approximated using the expression for unicity distance by Shannon [34]:

$$n_u \approx \left\lceil \frac{H(r)}{\rho} \right\rceil \tag{4.32}$$

The number of ciphertext blocks necessary to attempt effective ciphertext-only cryptanalysis for the i^{th} key triple is thus:

$$b_u = \left\lceil \frac{H(r)}{\rho c_i \log_2 N} \right\rceil \tag{4.33}$$

Assume the random selection has a uniform distribution. The probability of selecting any particular key is then $((N^{c_i})!)^{-1}$. Since there are $(N^{c_i})!$ keys in all, the entropy of the key space is given by

$$H(r_i) = -\sum_j p_j \log_2 p_j = \\ -(N^{c_i})! \cdot \frac{1}{(N^{c_i})!} \cdot -\log_2((N^{c_i})!) = \log_2((N^{c_i})!) \quad (4.34)$$

The key length in bits as a function of N and c_i is approximately:

$$l_{r_i}(N, c_i) = \lceil c_i \log_2 N \rceil N^{c_i}, \quad (4.35)$$

which is more than the entropy computed in equation 4.34. Thus there is some redundance in the key itself.

As with univariate encryption, it is possible, to some extent, to avoid frequency-based cryptanalysis by regularly re-encrypting f using the method outlined in section 4.6. Re-encryption intervals are computed by dividing b_u by the number of times the same key information is used in any one application of $E(f)$. A scheme such as this may work only if the data encrypted by $E(f)$ are:

1. re-encrypted, which raises the number of times the new key is applied, unless it is possible to assume that such a use is carried out within a trusted computing base;
2. thrown away, to reduce the number of times the new key is applied; or
3. archived along with the old key.

4.7.4 Known-plaintext Attack

Instead of being able to apply the keys, the cryptanalyst merely has access to a limited number (*ciphertext*,*plaintext*) pairs. If the cryptanalyst possesses N^{c_i} (*ciphertext*,*plaintext*) pairs for the i^{th} key triple in $K = \{c_i, r_i, s_i\}_{i=1}^{k}$, that key triple must be considered compromised (see corollary 27).

If the cryptanalyst possesses $C < N^{c_i}$ (*ciphertext,plaintext*) pairs for a given key pair, the cryptanalyst knows C of the N^{c_i} unknowns needed to find the keys. This gives the reduced unicity distance

$$b'_u = \left\lceil \frac{H(r_i)'}{\rho \log_2(N^{c_i} - C)} \right\rceil,$$

where $H(r_i)' = \log_2((N^{c_i} - C)!)$.

4.7.5 Functional Chosen-ciphertext Attack

The functional ciphertext is on the form in equation 4.26. Each encryption key may be written

$$r_{j,h}(x_1, \ldots, x_{c_j}) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_{c_j}=0}^{N-1} A_{j,h,i_1,\ldots,i_{c_j}} x_1^{i_1} \ldots x_{c_j}^{i_{c_j}}, \tag{4.36}$$

where $1 \leq h \leq c_j$. Decryption keys may be written

$$s_{j,h}(x_1, \ldots, x_{c_j}) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_{c_j}=0}^{N-1} B_{j,h,i_1,\ldots,i_{c_j}} x_1^{i_1} \ldots x_{c_j}^{i_{c_j}}, \tag{4.37}$$

where $1 \leq h \leq c_j$. The plaintext function may in general be written

$$f_j(x_1, \ldots, x_m) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_m=0}^{N-1} \phi_{j,i_1,\ldots,i_m} x_1^{i_1} \ldots x_m^{i_m}. \tag{4.38}$$

The functional ciphertext $E(f)$ may be written

$$E(f)_j = \sum_{i_1=0}^{N-1} \cdots \sum_{i_m=0}^{N-1} \Phi_{j,i_1,\ldots,i_m} x_1^{i_1} \ldots x_m^{i_m}. \tag{4.39}$$

An efficient attack is possible. The attack can also be used with polynomial representations, provided the generated functional ciphertexts are converted to polynomials prior to decryption.

THEOREM 32 *Fix a multivariate encryption template U and key triples $K = \{(c_i, r_i, s_i)\}_{i=1}^{k}$. If a cryptanalyst can generate plaintexts f from ciphertexts $E(f)$, that have been generated using U and K, and the cryptanalyst knows all the block sizes c_i, then a functional chosen-ciphertext attack can be completed in asymptotic time $\mathcal{O}(NnN^m)$, excluding any time spent doing polynomial interpolation or converting polynomials to function tables, which is nearly linear in the number of coefficients nN^m needed to represent f using an interpolation polynomial.*

PROOF: In this case, the keys and functions are represented using function tables. To reveal the keys, the constant mappings $(0, \ldots, 0), \ldots, (N-1, \ldots, N-1)$ are generated for each mapping block, and successively decrypted using U and K. For each block j set $(E(f)_{a+1}, \ldots, E(f)_{a+c_j}) = (v_1, \ldots, v_{c_j}) \in \mathbb{Z}_N^{c_j}$. It follows that

$$E(f)_h(r_{l+1}(\vec{w}_1), \ldots, r_k(\vec{w}_{k-l})) = v_{h-a} = E(f)_h,$$

further implying

$$\begin{aligned} f_h = (s_j(E(f)_{a+1}(r_{l+1}(\vec{w}_1), \ldots, r_k(\vec{w}_{k-l}), \ldots, \\ E(f)_{a+c_j}(r_{l+1}(\vec{w}_1), \ldots, r_k(\vec{w}_{k-l})))_{h-a} = s_{j,h}(v_1, \ldots, v_{c_j}). \end{aligned}$$

In this way it becomes possible to reconstruct the permutations $s_1, \ldots, s_l$ (and thus also $r_1, \ldots, r_l$) using radix sorting of the type described in the proof of lemma 11.

The next step is in principle similar to the corresponding part of the proof of theorem 18. Instead of constructing bands where individual variables vary, construct bands where individual blocks vary. The ideal case is described first, where $\sum_{i=l+1}^{k} c_i \leq \sum_{j=1}^{l} c_j$. In the following set $a =$

$\sum_{i=l+1}^{k} c_i$.

$$
\begin{array}{l}
E(f)_j(0,\ldots,0) = 0 \text{ for all } j \in \{1,\ldots,n\} \\
(E(f)_1(\underbrace{0,\ldots,0,1}_{\text{block } l+1},0,\ldots,0),\ldots,E(f)_{c_{l+1}}(\underbrace{0,\ldots,0,1}_{\text{block } l+1},0,\ldots,0)) = (\underbrace{0,\ldots,0,1}_{c_{l+1}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
(E(f)_1(\underbrace{N-1,\ldots,N-1}_{\text{block } l+1},0,\ldots,0),\ldots, \\
\quad E(f)_{c_{l+1}}(\underbrace{N-1,\ldots,N-1}_{\text{block } l+1},0,\ldots,0) = (\underbrace{N-1,\ldots,N-1}_{c_{l+1}}) \\
(E(f)_{c_{l+1}+1}(0,\ldots,0,\underbrace{0,\ldots,0,1}_{\text{block } l+2},0,\ldots,0),\ldots, \\
\quad E(f)_{c_{l+1}+c_{l+2}}(0,\ldots,0,\underbrace{0,\ldots,0}_{\text{block } l+2},0,\ldots,0)) = (\underbrace{0,\ldots,0,1}_{c_{l+2}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
(E(f)_{c_{l+1}+1}(0,\ldots,0,\underbrace{N-1,\ldots,N-1}_{\text{block } l+2},0,\ldots,0),\ldots, \\
\quad E(f)_{c_{l+1}+c_{l+2}}(0,\ldots,0,\underbrace{N-1,\ldots,N-1}_{\text{block } l+2},0,\ldots,0)) = (\underbrace{N-1,\ldots,N-1}_{c_{l+2}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
(E(f)_{a-c_k+1}(0,\ldots,0,\underbrace{0,\ldots,0,1}_{\text{block } k}),\ldots,\ E(f)_a(0,\ldots,0,\underbrace{0,\ldots,0,1}_{\text{block } k})) = (\underbrace{0,\ldots,0,1}_{c_k}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
(E(f)_{a-c_k+1}(0,\ldots,0,\underbrace{N-1,\ldots,N-1}_{\text{block } k}),\ldots, \\
E(f)_a(0,\ldots,0,\underbrace{N-1,\ldots,N-1}_{\text{block } k}) = (\underbrace{N-1,\ldots,N-1}_{c_k})
\end{array}
\tag{4.40}
$$

and $E(f)_j(\vec{x}) = 0$ otherwise. Note that all the arguments in equation 4.40 are encrypted. When the resulting $E(f)$ is decrypted to functional plain-

text, f will be defined as follows:

$$
\begin{aligned}
&(f_1(\underbrace{s_{l+1}(0,\ldots,0,1)}_{\text{block } l+1}, s_{l+2}(\vec{0}),\ldots,s_k(\vec{0})),\ldots,\\
&\ f_{c_{l+1}}(\underbrace{s_{l+1}(0,\ldots,0,1)}_{\text{block } l+1}, s_{l+2}(\vec{0}),\ldots,s_k(\vec{0}))) = \hat{s}_{l+1}(0,\ldots,0,1)\\
&\qquad\qquad\vdots\\
&(f_1(\underbrace{s_{l+1}(N-1,\ldots,N-1)}_{\text{block } l+1}, s_{l+2}(\vec{0}),\ldots,s_k(\vec{0})),\ldots,\\
&\ f_{c_{l+1}}(\underbrace{s_{l+1}(N-1,\ldots,N-1)}_{\text{block } l+1}, s_{l+2}(\vec{0}),\ldots,s_k(\vec{0}))) = \hat{s}_{l+1}(N-1,\ldots,N-1)\\
&(f_{c_{l+1}+1}(s_{l+1}(\vec{0}),\underbrace{s_{l+2}(0,\ldots,0,1)}_{\text{block } l+2}, s_{l+3}(\vec{0}),\ldots,s_k(\vec{0})),\ldots,\\
&\ f_{c_{l+1}+c_{l+2}}(s_{l+1}(\vec{0}),\underbrace{s_{l+2}(0,\ldots,0,1)}_{\text{block } l+2}, s_{l+3}(\vec{0}),\ldots,s_k(\vec{0}))) = \hat{s}_{l+2}(0,\ldots,0,1)\\
&\qquad\qquad\vdots\\
&(f_{c_{l+1}+1}(s_{l+1}(\vec{0}),\underbrace{s_{l+2}(N-1,\ldots,N-1)}_{\text{block } l+2}, s_{l+3}(\vec{0}),\ldots,s_k(\vec{0})),\ldots,\\
&\ f_{c_{l+1}+c_{l+2}}(s_{l+1}(\vec{0}),\underbrace{s_{l+2}(N-1,\ldots,N-1)}_{\text{block } l+2}, s_{l+3}(\vec{0}),\ldots,s_k(\vec{0})))\\
&= \hat{s}_{l+2}(N-1,\ldots,N-1)\\
&\qquad\qquad\vdots\\
&(f_{a-c_k+1}(s_{l+1}(\vec{0}),\ldots,s_{k-1}(\vec{0}),\underbrace{s_k(0,\ldots,0,1)}_{\text{block } k}),\ldots,\\
&\ f_a(s_{l+1}(\vec{0}),\ldots,s_{k-1}(\vec{0}),\underbrace{s_k(0,\ldots,0,1)}_{\text{block } k})) = \hat{s}_k(0,\ldots,0,1)\\
&\qquad\qquad\vdots\\
&(f_{a-c_k+1}(s_{l+1}(\vec{0}),\ldots,s_{k-1}(\vec{0}),\underbrace{s_k(N-1,\ldots,N-1)}_{\text{block } k}),\ldots,\\
&\ f_a(s_{l+1}(\vec{0}),\ldots,s_{k-1}(\vec{0}),\underbrace{s_k(N-1,\ldots,N-1)}_{\text{block } k})) = \hat{s}_k(N-1,\ldots,N-1),
\end{aligned}
\tag{4.41}
$$

where $\hat{s}_i : \mathbb{Z}_N^{c_i} \longrightarrow \mathbb{Z}_N^{c_i}$ for $l < i \leq k$. $\hat{s}$ is a notational convenience, and is defined as follows:

$$\hat{s}_{i,j} = s_{g,h}, \tag{4.42}$$

where $1 \leq g \leq l$ is such that $\sum_{d=1}^{g} c_d + h = \sum_{e=l+1}^{i} c_e + j$.

Since $s_1, \ldots, s_n$ are known, the right-hand side can be re-encrypted, generating pairs $(x_i, s_j(x_i))$ when $n < j \leq n + m$. Using radix sort again, sorted tables for the remaining keys can be constructed. If not all variable blocks can be fitted into one run of this attack, remaining variable blocks are attacked in subsequent similar runs, where instead of block $l + 1$, one begins with another variable block $a > l + 1$.

If $m > n$, but all $c_i < n$ for $l < i \leq k$, the attack above is repeated with different selections of variable blocks until all s_i are generated. If $c_i > n$ for some $l < i \leq k$, the attack is modified. A series of in all $\lceil c_i/n \rceil$ rounds generating data for blocks of n components at a time is needed. The first such round is shown in the equation 4.43.

$$\begin{aligned}
&E(f)_j(\vec{0}) = 0 \text{ for all } j \in \{1, \ldots, n\} \\
&(E(f)_1(0, \ldots, 0, \underbrace{\overbrace{0, \ldots, 0}^{\text{first } n}, \ldots, \overbrace{0, \ldots, 0, 1}^{\text{last } c_i \mod n}}_{\text{block } i}, 0, \ldots, 0), \ldots, \\
&\quad E(f)_n(0, \ldots, 0, \overbrace{0, \ldots, 0}^{\text{first } n}, \ldots, \overbrace{0, \ldots, 0, 1}^{\text{last } c_i \mod n}, 0, \ldots, 0)) = (\underbrace{0, \ldots, 0}_{n}) \\
&\qquad\qquad \vdots \\
&(E(f)_1(0, \ldots, 0, \underbrace{\overbrace{N-1, \ldots, N-1}^{\text{first } n}, \ldots, \overbrace{N-1, \ldots, N-1}^{\text{last } c_i \mod n}}_{\text{block } i}, 0, \ldots, 0), \ldots, \\
&\quad E(f)_n(0, \ldots, 0, \overbrace{N-1, \ldots, N-1}^{\text{first } n}, \ldots, \overbrace{N-1, \ldots, N-1}^{\text{last } c_i \mod n}, 0, \ldots, 0)) \\
&= (\underbrace{N-1, \ldots, N-1}_{n})
\end{aligned} \tag{4.43}$$

When all rounds are completed, sorted function tables for the s_i's can be generated using radix sort on the concatenated vectors.

Functions encryptable under U are defined by tables with N^{n+m} entries. Generating the tables for the first half of the attack takes $\mathcal{O}(nN^{m+1})$ time. Decrypting them also takes $\mathcal{O}(nN^{m+1})$ time. Generating the table(s) for the second half of the attack takes $\mathcal{O}(nN^m\lceil m/n\rceil)$ time. Decrypting it takes $\mathcal{O}(nN^m)$ time. Asymptotic time complexity is therefore roughly $\mathcal{O}(nN^{m+1})$, which is N times the number of coefficients needed to represent a interpolation polynomial over $\mathbb{Q}$ for any f encryptable under U $\square$.

Notice that the attack in the proof above only works if the cryptanalyst knows all c_i, or can find all c_i efficiently. Unfortunately, it turns out that knowledge of these block sizes is not sufficient to prevent a successful functional chosen-ciphertext attack.

THEOREM 33 *Fix a multivariate encryption template U and key triples $K = \{(c_i, r_i, s_i)\}_{i=1}^{k}$. If a cryptanalyst can generate plaintexts f as function tables from ciphertexts $E(f)$ represented as function tables, generated using U and K, then a functional chosen-ciphertext attack can be completed in asymptotic time at least*

$$\mathcal{O}(NnN^m + \frac{N(N^m-1)+N(N^n-1)}{N-1}),$$

and at most

$$\mathcal{O}(NnN^m + m(m+1)N^m + n(n+1)N^n)).$$

PROOF: The cryptanalyst carries out the same attack as in the proof of theorem 32 with some differences. All function components are collected into one block with block size $c_1' = n$. All variables are collected into one block with block size $c_2' = m$. The attack in the proof of theorem 32 is then carried out with these block sizes. The result is an encryption function r_1' for the function components, and a decryption function s_2' for the variables. This step takes $\mathcal{O}(NnN^m)$ operations, and is thus independent of the actual number of blocks and the actual block sizes employed in U.

The next step is to recover the real block sizes, and the real keys. The functions r'_1 and s'_2 are represented as function tables. Thus $\vec{x} = s'_2(\vec{y})$, $\vec{x}, \vec{y} \in \mathbb{Z}_N^m$. Set $c_{l+1} = 1$. Compare $s'_2(y_1, \ldots, y_{c_{l+1}}, 0, \ldots, 0)$ with $s'_2(y_1, \ldots, y_{c_{l+1}}, y_{c_{l+1}+1}, \ldots, y_m)$ for all other $N^{m-c_{l+1}} - 1$ possible combinations of $y_{c_{l+1}+1}, \ldots, y_m$. If there is no difference, $c_{l+1} = 1$, otherwise add one to c_{l+1}, and repeat until there no differences are found. Repeat for c_{l+2} and onwards, but ignoring the first c_{l+1} components, so that only every $N^{c_{l+1}}$th entry is used. The next round only every $N^{c_{l+1}+c_{l+2}}$nd entry is used, etc. This does not necessarily provide block sizes identical to those in U, but the block sizes so selected are effective and minimal. A similar procedure is followed to find block sizes for the function components.

If one uses the number of comparisons of individual vector components as the unit operation for time complexity purposes, at least

$$\sum_{i=1}^{m} N^i + \sum_{i=1}^{n} N^i = \frac{N(N^m - 1) + N(N^n - 1)}{N - 1}$$

such operations are needed to find the block sizes, and at most

$$\sum_{i=1}^{m} iN^m + \sum_{i=1}^{n} iN^n = \frac{1}{2}(m(m+1)N^m + n(n+1)N^n)$$

such operations are needed□.

4.7.6 Functional Chosen-plaintext Attack

The analysis is similar to that in section 4.7.5, and so is the result.

COROLLARY 34 *Multivariate encryption is vulnerable to functional chosen-plaintext attack.*

PROOF: Follows from theorem 33□.

4.7.7 Functional Ciphertext-only Attack

There are two obvious ways to attack a functional ciphertext (as with univariate encryption):

1. direct decomposition of each component of $E(f)$, and/or
2. accumulation of enough functional ciphertexts $E(f), E(g), \ldots$ to do a frequency-based attack to find the key.

The second alternative is practical if:

- the key pairs, along with the multivariate encryption template, do not vary with each encryption, so that accumulation of sufficient amounts of functional ciphertext encrypted with the same sets of key pairs is feasible;
- the plaintext language has a fair amount of redundancy, and
- the cryptanalyst is capable of recognizing the plaintext.

This analysis will concentrate on decomposition attacks. Only proper multivariate cases are considered, where at least one $c_i > 1$. Multivariate encryption is based on the assumption that decomposing $E(f)$ to find one or more of the mappings $f_1, \ldots, f_n$, $r_1, \ldots, r_k$, $s_1, \ldots, s_k$ is infeasible.

The functional ciphertext is on the form in equation 4.26. Consider the case $E(f) = (r_1, \ldots, r_l) \circ f$. It should be obvious that this case is neither covered by the Simple Multivariate Decomposition Problem (problem 9), nor the reduced version (problem 10). Decomposition of $E(f)$ is a problem covered at least in part by the General Polynomial Decomposition Problem.

PROBLEM 11 (GENERAL POLYNOMIAL DECOMPOSITION) *Given polynomial $f(x_1, \ldots, x_n) \in K[x_1, \ldots, x_n]$ and some subset of the following: polynomial $g(y_1, \ldots, y_m) \in K[y_1, \ldots, y_m]$, polynomials $h_1, \ldots, h_m \in K[x_1, \ldots, x_n]$, and templates specifying the form of polynomials g and $h_1, \ldots, h_m$, decide if there exists a functional decomposition $g, h_1, \ldots, h_m$ of f such that g and*

$h_1, \ldots, h_m$ are in the form specified by the template. If so, compute those coefficients of g and the h_i's which were not given.

Since a special case of problem 11, the S-1-decomposition problem, has been proven NP-hard, General Polynomial Decomposition is therefore also NP-hard (see [18]). Even though problem 11 is defined in a very general manner, I consider it necessary to emphasize the reduced case:

PROBLEM 12 (GENERAL REDUCED POLYNOMIAL DECOMPOSITION) *Given polynomial $f(\vec{x}) \in \hat{K}[\vec{x}]$, $\hat{K}$ an algebraic extension of a commutative ring K, $\vec{x} \in K^m$, and some subset of the following: a polynomial $g(\vec{z}) \in K[\vec{z}]$, $\vec{z} \in K^n$, polynomials $h_1(\vec{x}), \ldots, h_n(\vec{x}) \in K[\vec{x}]$, and templates specifying the form of the polynomials g and $h_1, \ldots, h_n$, decide if there exist polynomials $g, h_1, \ldots, h_n$ satisfying the template such that $g \circ (h_1, \ldots, h_n)$ is reducible to f. If so, compute those coefficients of g and the h_is which were not given.*

Problem 11 has been proven NP-hard. Since problem 12 is, strictly speaking a special case of problem 11 the way Dickerson has expressed it, I conjecture the following:

CONJECTURE 35 *General Reduced Polynomial Decomposition is at least as hard as General Polynomial Decomposition, so it is NP-hard.*

As with the univariate case, it is possible to construct a system of equations that the coefficients of the reduced polynomial must satisfy. The following relation holds for all j such that :

$$\begin{aligned}\Phi_{j,i_1,\ldots,i_m} \equiv \psi_{j,\vec{r}}(&A_{a,1,0,\ldots,0}, \ldots, A_{a,c_a,N-1,\ldots,N-1},\\ &B_{l+1,1,0,\ldots,0}, \ldots, B_{l+1,c_{l+1},N-1,\ldots,N-1}, \ldots,\\ &B_{k,1,0,\ldots,0}, \ldots, B_{k,c_k,N-1,\ldots,N-1},\\ &\phi_{j,0,\ldots,0}, \ldots, \phi_{j,N-1,\ldots,N-1})(\mod N),\end{aligned} \tag{4.44}$$

where a is a positive integer such that $\sum_{i=1}^{a} c_i < j \leq \sum_{i=1}^{a+1} c_i$. Set $q = \sum_{i=1}^{a} c_i$. Each $\psi_{j,\vec{r}}$ is the general expression for the reduced composition of r_a with $f_{q+1}, \ldots, f_{q+c_a}$, which in turn may have one or more of its variables

substituted by the corresponding s_h. There are thus nN^m equations with $nN^m + \sum_{i=1}^{k} c_i N^{c_i}$ independent unknowns.

LEMMA 36 *Given a functional composition $E(f) : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ with coefficients $\Phi_{j,\vec{\iota}}$ satisfying equation 4.44, and $N \geq 6$, there does not exist any analytic method of solving the system of equations resulting from the relation in equation 4.44.*

PROOF: Similar to that of lemma 23 □.

4.7.8 Functional Known-Plaintext Attack

The cryptanalyst has access to a limited number of (*functional ciphertext*, *functional plaintext*) pairs.

LEMMA 37 *Fix a multivariate encryption template U and a set of key triples $K = \{(c_i, r_i, s_i)\}_{i=1}^{k}$. If the cryptanalyst knows one (*functional ciphertext*, functional plaintext*) pair for U and K, then any other functional ciphertext $E(f)$ generated using U and K is compromised, provided*

$$\sum_{i=1}^{k} c_i N^{c_i} \leq nN^m.$$

PROOF: The knowledge of the single pair gives knowledge of the coefficients $\Phi_{j,\vec{\iota}}$ and $\phi_{j,\vec{\iota}}$ for all $j \in \{1, \ldots, n\}$ and all $\vec{\iota} \in \mathbb{Z}_N^m$. Setting up the relations given by equation 4.44 results in a system of up to nN^m equations with $\sum_{i=1}^{k} c_i N^{c_i}$ independent unknowns and $2\sum_{i=1}^{k} c_i N^{c_i}$ dependent unknowns. If the number of independent unknowns, $\sum_{i=1}^{k} c_i N^{c_i}$, still exceeds the number of equations, nN^m, the keys are not compromised □.

4.8 Parametric Encryption

This is a generalization of multivariate encryption, which is designed primarily with the register machine in mind. It is an attempt at using a public

randomizer with respect to effective key choice. One consequence is that this cryptosystem in general does not appear to support key re-encryption. Nor does decryption of the functional ciphertexts appear to be possible in general for parametric encryption templates that do not reduce to multivariate or univariate encryption templates. Parametric encryption uses key quadruples (c_i, g_i, r_i, s_i), where the new element, relative to multivariate encryption, indicates a variable block that is to be taken as parameter in an encryption/decryption.

Fix an integer $N > 1$, positive integers m and n, a positive integer $k \leq n + m$, and a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. Select a sequence of integers $\{c_i\}_{i=1}^k$ such that $\sum_{i=1}^k c_i = n + m$ and $\sum_{i=1}^l c_i = n$ for some $l < k$. Select another sequence of integers $\{g_i\}_{i=1}^k$ such that $l < g_i \leq k$ or $g_i < l$. The series of integers c_i represents "block sizes" used in encryption and decryption. The series of integers g_i indicates a variable block to be used in the parametrization of the i^{th} key quadruple, or, if less than l, indicates that no such parametrization takes place. If $N = P^e$, where P is a prime, and e a positive integer, mappings may either be represented as function tables or interpolation polynomials over $\mathbb{Z}_N$. If $e = 1$, $\mathbb{Z}_N$ is interpreted as the integers modulo N. If $e > 1$ they are interpreted as polynomials in $\mathbb{Z}_P[x]/p(x)$, where $p(x)$ is an irreducible degree e polynomial over $\mathbb{Z}_P$. Otherwise, all mappings are assumed to be represented using function tables. The variables of f are grouped into blocks of $c_{l+1}, c_{l+2}, \ldots, c_k$ variables as follows

$$\underbrace{x_1, \ldots, x_{c_{l+1}}}_{\text{block } l+1}, \underbrace{x_{c_{l+1}+1}, \ldots, x_{c_{l+1}+c_{l+2}}}_{\text{block } l+2}, \ldots, \underbrace{x_{m-c_k+1}, \ldots, x_m}_{\text{block } k}. \tag{4.45}$$

As before, the i^{th} variable block is written $\vec{w}_i$.

The components of f are grouped into blocks of c_1-, c_2-,. . . , c_l-component mappings as follows:

$$(\underbrace{f_1, \ldots, f_{c_1}}_{\text{block 1}}, \underbrace{f_{c_1+1}, \ldots, f_{c_1+c_2}}_{\text{block 2}}, \ldots, \underbrace{f_{n-c_l+1}, \ldots, f_n}_{\text{block l}}). \tag{4.46}$$

This is identical to the grouping done for multivariate encryption, so example 3 is also illustrative in this context.

The assignment of parameter groups is perhaps best illustrated by an example.

EXAMPLE 5 *Continuing example 3, the function component grouping indicated in equation 4.23 along with the variable component grouping indicated in equation 4.25 could have as two possibilities the parameter groups*

$$(\underbrace{\overbrace{x_1,x_2}^{g_1=3}}_{\text{block 1}}\underbrace{f_1,f_2,f_3}, \underbrace{\overbrace{f_4,f_5}^{g_2=0}}_{\text{block 2}}), \underbrace{\overbrace{x_3,x_4}^{g_3=4}}_{\text{block 3}}, \underbrace{\overbrace{x_3,x_4}^{g_4=0}}_{\text{block 4}}. \tag{4.47}$$

or the parameter groups

$$(\underbrace{\overbrace{x_1,x_2}^{g_1=3}}_{\text{block 1}}\underbrace{f_1,f_2,f_3}, \underbrace{\overbrace{f_4,f_5}^{g_2=0}}_{\text{block 2}}), \underbrace{\overbrace{x_3,x_4}^{g_3=4}}_{\text{block 3}}, \underbrace{\overbrace{x_3,x_4}^{g_4=3}}_{\text{block 4}}. \tag{4.48}$$

More combinations are possible, and by setting all g_i to 0 or a number not in $\{l+1,\ldots,k\}$, parametric encryption is reduced to multivariate encryption. Also, by selecting only blocks containing one function component or one variable, parametric univariate encryption is possible as a special case.

Writing a general expression for the composition operations producing $E(f)$ requires a trick. Define $a_i = \sum_{j=l+1}^{g_i-1} c_j$, and $\vec{v}_i = (x_{a_i+1},\ldots,x_{a_i+c_{g_i}})$ if $g_i \in \{l+1,\ldots,k\}$, and $\vec{v}_i = (x_1,\ldots,x_{c_{g_i}})$ otherwise. Define

$$\dot{r}_i(y_1,\ldots,y_n,\vec{v}_i) = \begin{cases} r_i(y_1,\ldots,y_n), & g_i \notin \{l+1,\ldots,k\} \\ r_i(y_1,\ldots,y_n,\vec{v}_i), & g_i \in \{l+1,\ldots,k\} \end{cases}, \tag{4.49}$$

and define

$$\dot{s}_i(y_1,\ldots,y_n,\vec{v}_i) = \begin{cases} s_i(y_1,\ldots,y_n), & g_i \notin \{l+1,\ldots,k\} \\ s_i(y_1,\ldots,y_n,\vec{v}_i), & g_i \in \{l+1,\ldots,k\} \end{cases}. \tag{4.50}$$

The general expression for the parametric encryption of f may then be written:

$$\begin{aligned} E(f) = (&\dot{r}_1(f_1(\dot{s}_{l+1}(\vec{w}_1,\vec{v}_{l+1}),\ldots,\dot{s}_k(\vec{w}_{k-l},\vec{v}_k)),\vec{v}_1),\ldots, \\ &\dot{r}_l(f_l(\dot{s}_{l+1}(\vec{w}_1,\vec{v}_{l+1}),\ldots,\dot{s}_k(\vec{w}_{k-l},\vec{v}_k)),\vec{v}_l)). \end{aligned} \tag{4.51}$$

Some examples should illustrate what such compositions could look like.

EXAMPLE 6 *Assume we are given the variable and function component groupings of example 5, along with the corresponding parametrizations. Using the parametrization in equation 4.47, the expression for* $E(f)$ *is*

$$\begin{aligned} E(f) = r_1(&f_1(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2)),\\ &f_2(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2)), f_3(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2)), \vec{w}_1),\\ &r_2(f_4(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2)), f_5(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2))). \end{aligned} \quad (4.52)$$

Using the parametrization given in equation 4.48 results in the expression

$$\begin{aligned} E(f) = r_1(&f_1(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2, \vec{w}_1)),\\ &f_2(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2, \vec{w}_1)), f_3(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2, \vec{w}_1)), \vec{w}_1),\\ &r_2(f_4(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2, \vec{w}_1)), f_5(s_3(\vec{w}_1, \vec{w}_2), s_4(\vec{w}_2, \vec{w}_1))). \end{aligned} \quad (4.53)$$

Before encrypting f, one needs to

- define the sizes of the variable blocks;
- define the sizes of the function component blocks;
- define any parameters used by the keys of a given block;
- select the subset of variable blocks to be encrypted, by selecting a subset $I \subseteq \{l+1, \ldots, k\}$ of the variable block indexes; and
- select the subset of mapping component blocks to be encrypted, by selecting a subset $J \subseteq \{1, \ldots, l\}$ of the component block indexes.

Denote by F_j the j^{th} mapping component block $f_{a+1}, \ldots, f_{a+c_j}$, where $a = \sum_{i=1}^{j-1} c_i$. Since block sizes may vary, keys are defined by quadruples, and for the sake of simplicity, these quadruples will henceforth be referred to as "key quadruples". Key quadruples whose indices are not in $I \cup J$ are defined to be the identity mapping. The remaining key triples with indices in $I \cup J$ may or may not be restricted in some manner, as for univariate encryption. This allows the tailoring of the encrypted mapping to:

- take specified blocks of inputs in ciphertext and other blocks in plaintext, and
- produce specified blocks of outputs in ciphertext and other blocks in plaintext.

DEFINITION 13 (PARAMETRIC ENCRYPTION TEMPLATE) *A tuple* $(m, n, N, I, J, k, \{c_i\}_{i=1}^{k}, \{g_i\}_{i=1}^{k})$ *where:*

- m *is the arity (number of variables) of the mappings it applies to,*
- n *is the number of components of the mappings it applies to,*
- N *is the order of the ring over which the mappings it applies to are defined,*
- $I \subseteq \{l+1, \ldots, k\}$ *contains the indices of blocks of variables to be used in encrypted form,* l *is such that* $\sum_{i=1}^{l} c_i = n$*,*
- $J \subseteq \{1, \ldots, l\}$ *contains the indices of the blocks of mapping components to be encrypted,*
- k *is the total number of blocks defined,*
- $\{c_i\}_{i=1}^{k}$ *are the block sizes, and*
- $\{g_i\}_{i=1}^{k}$ *indicate the variable block to be used as parameter for the* i^{th} *block, if any.*

Once all constraints on the key quadruples have been defined, the encryption functions may be generated. The encryption functions can be considered indexed families of permutations. If $l < g_i \leq k$, the mappings $r_i, s_i : \mathbb{Z}_N^{c_i + c_{g_i}} \longrightarrow \mathbb{Z}_N^{c_i}$, are generated. The mapping r_i is generated such that if the last c_{g_i} variables are fixed, r_i is a permutation of $\mathbb{Z}_N^{c_i}$, and s_i its inverse. Thus the parameter effectively selects one of many permutations to use in the evaluation of $E(f)$. If $g_i \leq l$ or $k < g_i$, the permutations $r_i, s_i : \mathbb{Z}_N^{c_i} \longrightarrow \mathbb{Z}_N^{c_i}$, are generated.

4.9 Cryptanalysis of Parametric Encryption

Parametric encryption is characterized by the following:

1. It is an asymmetric *secret* key algorithm—neither of the keys r_i, s_i in a keyquadruple (c_i, g_i, r_i, s_i) may be publicized, as knowing one allows the construction of the other by interpolation or radix sort. Furthermore, g_i should be kept secret, as it may significantly hinder effective cryptanalysis. Keeping c_i secret may also make cryptanalysis harder.

2. For every set of key quadruples and encryption generated with the key quadruples, there are *two types* of plaintext and ciphertext:

   (a) The mapping f represents one of the types of plaintext, and $E(f)$ its encrypted equivalent.

   (b) The datum $\vec{x}$ represents the other type of plaintext, and $(E(f))(\vec{x})$ its *transformed* equivalent. Note that $(E(f))(\vec{x})$ is not necessarily encrypted, a fact which is not necessarily known by the attacker.

3. Decryption appears to be infeasible for all practical intents and purposes.

4. The encryption key actually applied to any variable or component block may in part be selected by that block's parameter block.

LEMMA 38 *Fix a parametric encryption template U. If the cryptanalyst has the function table representation for r_i or s_i, but not both, for the i^{th} key tuple, the cryptanalyst can find the tabular representation for s_i or r_i, respectively, in time $\mathcal{O}(N^{c_i})$ if r_i has no parameter and in time $\mathcal{O}(N^{c_i+c_{g_i}})$ if r_i does.*

PROOF: Without loss of generality, assume the cryptanalyst knows r_i. There are two main cases:

CASE 1: The simplest case is when r_i has no parameter. The proof for this case is the same as the proof of lemma 26.

CASE 2: r_i has a parameter. If r_i is represented as a function table, it can be considered a list of pairs $((\vec{w}, \vec{v}_i), r_i(\vec{w}, \vec{v}_i))$, where $\vec{w} \in \mathbb{Z}_N^{c_i}$, $\vec{v}_i \in \mathbb{Z}_N^{c_{g_i}}$, and $r_i(\vec{w}, \vec{v}_i) \in \mathbb{Z}_N^{c_i}$. An ordered table for s_i can be constructed in time linear in $N^{c_i+c_{g_i}}$ using radix sort. For every $((\vec{w}, \vec{v}_i), r_i(\vec{w}, \vec{v}_i))$ set $\vec{w}' = r_i(\vec{w}, \vec{v}_i)$ and $s_i(\vec{w}', \vec{v}_i) = \vec{w}$ $\square$.

COROLLARY 39 *Fix a parametric encryption template U. If the cryptanalyst has $N^{c_i+c_{g_i}}$ distinct (*ciphertext*,* plaintext*) pairs for the same encryption keys that use a parameter, the cryptanalyst can construct a table for both the relevant encryption and decryption keys, r_i and s_i, in time $\mathcal{O}(N^{c_i+c_{g_i}})$.*

PROOF: r_i and s_i permute $\mathbb{Z}_N^{c_i}$ for one fixed parameter value, of which there are $N^{c_{g_i}}$. For every value of the parameter, the tabular representations of r_i and s_i can be constructed using radix sort. Each such construction takes time proportional to N^{c_i}, and each such construction must be repeated $N^{c_{g_i}}$ times to cover all the parameter values $\square$.

Corollary 39 doesn't cover the case where r_i has no parameter, as this is already covered by corollary 27.

There exist $(N^n)^{(N^m)}$ different mappings $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. With respect to key selection, parametric encryption differs somewhat from multivariate and univariate encryption. Each block i may or may not have another block assigned to it as parameter. If a block j is assigned (variable) block j as parameter, and block i is to have a permutation applied to it, then each value for block j selects one of N^{c_j} possible permutations to apply to block i. For large enough c_j relative to c_i, it is possible that more than one value of block j refer to the same permutation. There are $(N^{c_i})!$ ways of permutating block i, and N^{c_j} possible selections that block j can offer. Therefore, there are in general

$$\prod_{i=1}^{k} (N^{c_i})!\alpha(i), \tag{4.54}$$

ways of selecting keys for a given parametric encryption template, where

$$\alpha(i) = \begin{cases} 1, & g_i \leq l \text{ or } g_i > k \\ N^{c_{g_i}}, & l < g_i \leq k. \end{cases} \tag{4.55}$$

If no block has a variable block selected as parameter, this system reduces to multivariate encryption. A parametric encryption has the possibility of being perfectly secure if the expression in equation 4.54 exceeds $(N^n)^{(N^m)}$, the number of mappings encryptable by the template.

The rest of this cryptanalysis proceeds as for the previous two cryptosystems, in the hopes that comparisons between them become easier.

4.9.1 Chosen-ciphertext Attack

Chosen-ciphertext attacks require the cryptanalyst to be capable of using the key(s) directly.

As with uni- and multivariate encryption, it is possible to generate (*ciphertext*, *plaintext*) pairs, which thereafter can be used to generate both keys.

THEOREM 40 *Fix a parametric encryption template U. If a cryptanalyst can apply, but not read either r_i or s_i for a selected subset $A \subseteq K$ of all key quadruples $K = \{(c_i, g_i, r_i, s_i)\}_{i=1}^{k}$ applied to encrypt f using multivariate encryption to give $E(f)$, then:*

1. *all the keys in A are vulnerable,*
2. *any non-functional ciphertext encrypted by a key in A, input to- or output by $E(f)$ is vulnerable, and*
3. *any components of $E(f)$ encrypted using keys in A are vulnerable, provided the cryptanalyst also knows g_i for each key in A.*

PROOF: Without loss of generality, consider only the i^{th} key quadruple. If $g_i \leq l$ or $g > k$ (r_i and s_i do not take any parameter), then theorem 29 applies. Therefore assume that r_i and s_i take a parameter.

CLAIM 1: The cryptanalyst can now generate pairs $((\vec{w}_j, \vec{v}_i), r_i(\vec{w}_j, \vec{v}_i))$, which can be used to generate a function table for r_i. This can be done for every key belonging to a key quadruple in A.

CLAIM 2: Follows trivially from the proof of claim 1.

CLAIM 3: If the cryptanalyst has or generates a function table for $E(f)$ and the keys in A, then it is possible to recover part of f from $E(f)$ by symbolically substituting $\vec{w}_i$ with $r_{l+i}(\vec{w}_i, \vec{v}_{l+i})$ for all i, $1 \leq i \leq k - l$. Note that to do the substitution correctly, one must know where in $\vec{x}$ one finds the parameter values, so that it is possible to compute the correct substitution. Next s_i is composed with the i^{th} component block for all i, $1 \leq i \leq l$. The result, depending on A is one or more partially or fully decrypted function blocks of $E(f)$ $\square$.

COROLLARY 41 *Data encrypted by parametric encryption is vulnerable to chosen-ciphertext attacks.*

PROOF: Follows from theorem 40 $\square$.

COROLLARY 42 *Functional ciphertext encrypted by parametric encryption is vulnerable to chosen-ciphertext attacks only if the cryptanalyst knows what the parameters are.*

PROOF: Follows from theorem 40 $\square$.

4.9.2 Chosen-plaintext Attack

The results for this type of attack are very similar to those presented in subsection 4.9.1.

COROLLARY 43 *Data encrypted by parametric encryption is vulnerable to chosen-plaintext attacks.*

PROOF: Follows from theorem 40 $\square$.

COROLLARY 44 *Functional ciphertext encrypted by parametric encryption is vulnerable to chosen-plaintext attacks only if the cryptanalyst knows what the parameters are.*

PROOF: Follows from theorem 40 $\square$.

4.9.3 Ciphertext-only Attack

The encryption permutations r_i viewed with respect to data encryption, are in a sense non-periodic polyalphabetic substitution ciphers based on an N^{c_i}-letter alphabet. Assume for the rest of this section that block i makes use of a parameter. The case where block i has no parameter is covered by the analysis in section 4.7.3. Ciphertext-only attack is only possible if there is any detectable redundancy or pattern in $\vec{x}$ as a whole. Note, however, that a cryptanalyst may not know whether or not any particular block is encrypted with a parameter or not. In fact, the cryptanalyst may not even know how the blocks are defined.

The number of ciphertext bits necessary to attempt effective cryptanalysis of block number i can be approximated using the expression for the expected unicity distance by Shannon [34]:

$$n_u \approx \left\lceil \frac{H(r)}{\rho} \right\rceil . \tag{4.56}$$

As noted before, block i is encrypted with up to $A = \min\{(N^{c_i})!, N^{c_{g_i}}\}$ different alphabets.

The number of ciphertext blocks necessary to attempt effective ciphertext-only cryptanalysis is thus:

$$b_u = \left\lceil \frac{H(r)}{\rho(c_i \log_2 N + \log_2 A)} \right\rceil \tag{4.57}$$

Assuming the random selection has a uniform distribution, and that the different substitution ciphers are selected cyclicly, the entropy of the key space is given by

$$H(r) = A(N^{c_i})! \cdot \frac{1}{A(N^{c_i})!} \cdot -\log_2 \left(\frac{1}{A(N^{c_i})!} \right) = \log_2(A(N^{c_i})!) \tag{4.58}$$

The effective key length in bits as a function of N, c_i, and g_i is approximately:

$$l_r(N, c_i, g_i) = \lceil c_i \log_2 N \rceil A \tag{4.59}$$

which is still less than the minimum total number of bits needed to represent the key:

$$\lceil c_i \log_2 N \rceil N^{c_i + c_{g_i}}. \tag{4.60}$$

4.9.4 Known-plaintext Attack

The cryptanalyst has access to a limited number (*ciphertext*, *plaintext*) pairs. If a given block is not encrypted with a parameter, this reduces to the multivariate case. Assume therefore that the block i under consideration has been encrypted with a parameter.

Knowledge of the (*ciphertext*, *plaintext*) pairs can produce two initial contributions for the cryptanalyst:

- a reduced effective unicity distance, and
- if there are more than N^{c_i} pairs, conclusive evidence that a proper parametric encryption has been applied to the block in question.

4.9.5 Functional Chosen-ciphertext Attack

The functional ciphertext is on the form given in equation 4.51. Each encryption key that takes a parameter, may be written

$$r_{j,h}(x_1, \ldots, x_{c_j}, x_{c_j+1}, \ldots, x_{c_j+c_{g_j}}) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_{c_j+c_{g_j}}=0}^{N-1} A_{j,h,i_1,\ldots,i_{c_j+c_{g_j}}} x_1^{i_1} \ldots x_{c_j+c_{g_j}}^{i_{c_j+c_{g_j}}}, \tag{4.61}$$

where $1 \leq h \leq c_j$. Encryption keys that do not take a parameter may be written on the form given in equation 4.36. Decryption keys that take a parameter, may be written

$$s_{j,h}(x_1, \ldots, x_{c_j}, x_{c_j+1}, \ldots, x_{c_j+c_{g_j}}) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_{c_j+c_{g_j}}=0}^{N-1} B_{j,h,i_1,\ldots,i_{c_j+c_{g_j}}} x_1^{i_1} \ldots x_{c_j+c_{g_j}}^{i_{c_j+c_{g_j}}}, \tag{4.62}$$

where $1 \leq h \leq c_j$. Decryption keys that do not take a parameter may be written on the form given in equation 4.37. Similarly, the plaintext function may in general be written

$$f_j(x_1, \ldots, x_m) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_m=0}^{N-1} \phi_{j,i_1,\ldots,i_m} x_1^{i_1} \ldots x_m^{i_m}. \tag{4.63}$$

The functional ciphertext $E(f)$ may be written

$$E(f)_j = \sum_{i_1=0}^{N-1} \cdots \sum_{i_m=0}^{N-1} \Phi_{j,i_1,\ldots,i_m} x_1^{i_1} \ldots x_m^{i_m}. \tag{4.64}$$

The cryptanalyst does not know which keys take parameter blocks. This is an additional barrier to cryptanalysis.

THEOREM 45 *Fix a parametric encryption template U and key quadruples $K = \{(c_i, g_i, r_i, s_i)\}_{i=1}^{k}$. If a cryptanalyst can generate plaintexts f as function tables from ciphertexts $E(f)$, represented as function tables, generated using U and K, then the cryptanalyst can construct two pairs of functions (r_1', s_1') and (r_2', s_2') which allow bulk decryption of $E(f)$'s inputs and outputs.*

PROOF: The attack is identical to the first half of the attack described in the proof of theorem 33 □.

It is possible to employ the same algorithm to find block sizes for a functional ciphertext encrypted with parametric encryption. The dependencies introduced by parametrization, however, mean that many of the blocks arrived at will at best be large, and misleading.

COROLLARY 46 *Parametric encryption is vulnerable to functional chosen-ciphertext attack.*

PROOF: Follows from theorem 45 □.

4.9.6 Functional Chosen-plaintext Attack

The result here is identical to that in section 4.9.5.

COROLLARY 47 *Parametric encryption is vulnerable to functional chosen-plaintext attack.*

PROOF: Follows from theorem 45 □.

4.9.7 Functional Ciphertext-only Attack

There are two obvious ways to attack a functional ciphertext (as with univariate encryption):

1. direct decomposition of each component of $E(f)$, and/or
2. accumulation of enough functional ciphertexts $E(f), E(g), \ldots$ to do a frequency-based attack to find the key.

The second alternative is practical if:

- the key pairs, along with the parametric encryption template, do not vary with each encryption, so that accumulation of sufficient amounts of functional ciphertext encrypted with the same sets of key pairs is feasible;
- the plaintext language has a fair amount of redundancy, and
- the cryptanalyst is capable of recognizing the plaintext.

This analysis will concentrate on decomposition attacks. Only proper parametric cases are considered, where at least one $c_i > 1$. Parametric encryption is based on the assumption that decomposing $E(f)$ to find one or more of the mappings $f_1, \ldots, f_n$, $r_1, \ldots, r_k$, $s_1, \ldots, s_k$ is infeasible.

The univariate decomposition problems (7 and 8) are obviously not applicable at all. Neither are the simple multivariate decomposition problems (9 and 10) applicable. Only the general polynomial decomposition variants are applicable (problems 11 and 12) of those considered so far. Therefore conjecture 35 also applies here.

LEMMA 48 *Given a functional composition $E(f) : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ with coefficients $\Phi_{j,\vec{\iota}}$ satisfying equation 4.44 but with key constants on the form given in equations 4.61 and 4.62, and at least one key with degree ≥ 5, there does not exist any analytic method of solving the system of equations resulting from the relation in equation 4.44.*

PROOF: Similar to that of lemma 23 □.

4.9.8 Functional Known-Plaintext Attack

The cryptanalyst has access to a limited number of (*functional ciphertext*, *functional plaintext*) pairs.

CONJECTURE 49 *Fix a parametric encryption template U and a set of key quadruples $K = \{(c_i, g_i, r_i, s_i)\}_{i=1}^k$. If the cryptanalyst knows one (*functional ciphertext,functional plaintext*) pair for U and K, then any other functional ciphertext $E(f)$ is still secure, and neither U nor K is compromised.*

Because the cryptanalyst does not know g_i for any block, the cryptanalyst does not even know whether the ciphertext has been encrypted using proper parametric encryption or multivariate encryption. Thus the cryptanalyst does not even know the form of the equations relevant to the functional ciphertext. Even if this is known, the fact that the g_i are unknown mean that the cryptanalyst still cannot construct the correct equations. Thus it appears to be the case that parametric encryption resists functional known-plaintext attack.

Chapter 5

Computing with Encrypted Automata

Chapter 2 defined three different automata, along with methods of converting them to computing endomorphisms amenable to encryption. Chapter 3 demonstrated how the automata can be used with a simple device called the Turing platform to acheive (Turing) universal computation by simulating a universal Turing machine on the tape of the Turing platform. This chapter applies the encryption methods defined in chapter 4 to encrypt the automata representations of chapter 2. This allows, in theory, universal encrypted (Turing) computation.

To facilitate concise discussion, some more concepts are needed.

DEFINITION 14 (RUN) *A* run *of a mapping f is a series $R = \{(\vec{x}(i), \vec{z}(i), \vec{y}(i))\}_{i=0}^{L}$ that satisfies relations given in equations 3.5–3.7. Furthermore, $L \in \mathbb{N} \cup \{\omega\}$, and ω is the first infinite ordinal (the cardinality of $\mathbb{N}$).*

DEFINITION 15 (EQUIVALENCE UP TO ENCRYPTION (AUTOMATA)) *A run $R_1 = \{(\vec{x}(i), \vec{z}(i), \vec{y}(i))\}_{i=0}^{\kappa}$, is* equivalent up to encryption under U and K *with another run $R_2 = \{(\vec{X}(i), \vec{Y}(i), \vec{Z}(i))\}_{i=0}^{\kappa}$ if there exists:*

1. *a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ such that R_1 is a run of f,*

2. *an encryption template U and a collection of key-tuples K such that R_2 can be generated from R_1 as follows, denoting by S the dimension of $\vec{x}(i)$, O the dimension of $\vec{z}(i)$, and I the dimension of $\vec{y}(i)$:*

   (a) *U is a univariate encryption template:*

      i. *apply $r_1, \ldots, r_S$ to the respective components of each $\vec{x}(i)$ to produce $\vec{X}(i)$;*
      ii. *apply $r_{S+1}, \ldots, r_{S+O}$ to the respective components of each $\vec{z}(i)$ to produce $\vec{Z}(i)$; and*
      iii. *apply $r_{n+S+1}, \ldots, r_{n+S+I}$ to the respective components of each $\vec{y}(i)$ to produce $\vec{Z}(i)$.*

   (b) *U is a multivariate or parametric encryption template:*

      i. *find a such that $\sum_{i=1}^{a} c_i = S$;*
      ii. *apply $r_1, \ldots, r_a$ to the respective blocks of each $\vec{x}(i)$ to produce $\vec{X}(i)$;*
      iii. *apply $r_{a+1}, \ldots, r_l$ to the respective blocks of each $\vec{z}(i)$ to produce $\vec{Z}(i)$; and*
      iv. *apply $r_{l+a+1}, \ldots, r_k$ to the respective blocks of each $\vec{y}(i)$ to produce $\vec{Y}(i)$;*

   (c) *U is a parametric encryption template:*

      i. *find a such that $\sum_{i=1}^{a} c_i = S$;*
      ii. *apply $r_1, \ldots, r_a$ to the respective blocks of each $\vec{x}(i)$ and to the respective parameter blocks in $\vec{X}(i)$ to produce $\vec{X}(i)$;*
      iii. *apply $r_{a+1}, \ldots, r_l$ to the respective blocks of each $\vec{z}(i)$ and to the respective parameter blocks in $\vec{X}(i)$ to produce $\vec{Z}(i)$; and*
      iv. *apply $r_{l+a+1}, \ldots, r_k$ to the respective blocks of each $\vec{y}(i)$ and to the respective parameter blocks in $\vec{X}(i)$ to produce $\vec{Y}(i)$.*

Thus there is equivalence up to encryption between two runs if the (possibly partially) encrypted run R generated by $E(H)$ can be decrypted to another run R' of H.

DEFINITION 16 (COMPUTATIONAL EQUIVALENCE) *A functional ciphertext $E(f)$ is* computationally equivalent *to a functional plaintext f iff for every run R of f, there exists a unique run R' of $E(f)$ which is equivalent up to encryption.*

DEFINITION 17 (COMPUTATIONAL EQUIVALENCE FOR (M,T) PAIRS) *A pair $(E(M),T)$ such that M is an automaton that reads all its input from T's tape, and T a compatible Turing platform is* computationally equivalent *to a pair (M,T) if:*

1. *the finite control of M is computationally equivalent to $E(M)$ for some encryption template U and set of key tuples K generated using U; and*
2. *U is such that the key tuple applied to the output component of M dedicated to outputting the movement direction to T is set to the identity mapping;*
3. *for all computation steps a, $C(i,a)' = E(C(i,a))$, where:*
   - *$C(i,a)'$ is the value stored in cell i after a computation steps on the tape of T during $E(M)$'s computation,*
   - *$C(i,a)$ is the value stored in cell i after a computation steps on the tape of T during M's computation,*
   - *$E(C(i,a))$ is the encryption of $C(i,a)$ using the keys assigned to encrypting the output alphabet ($r_{S+1},\ldots,r_{n-1}$ for univariate encryption, $r_{b+1},\ldots,r_{l-1}$ for multivariate or parametric encryption (b being the last block of state components).*

5.1 Univariate Encryption of Programs

Section 4.2, describes how to encrypt a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. This section describes how univariate encryption is applied to computing endomorphisms and register automata to achieve encrypted computation.

Fix an automaton M (either a Mealy machine, BSS' automaton, or register automaton) with a representation H. H is either a polynomial mapping, or a mapping defined using a function table. M may or may not be augmented and/or obfuscated.

If M is a Mealy machine, H is the mapping (δ, λ). If M is a BSS' automaton, H is the computing endomorphism given in equation 2.16. If M is a register automaton, there is no directly corresponding computing endomorphism as with Mealy machines and BSS' automata, but rather a series of evaluations of encrypted mappings.

Recall that the general form of H for Mealy and BSS' automata is:

$$H : \mathbb{Z}_N^S \times \mathbb{Z}_N^I \longrightarrow \mathbb{Z}_N^S \times \mathbb{Z}_N^O. \tag{5.1}$$

5.1.1 M Is a Mealy Or BSS' Machine

The representation H is the mapping (δ, λ). Encrypting this mapping such that it still can carry out the original computation is done as follows:

1. Select the state components that are to be stored in encrypted form. Each state component stored in encrypted form must be decrypted and encrypted by the same key for each computation step. Thus if x_i, $1 \leq i \leq S$ is encrypted, one must have $(r_i, s_i) = (r_{n+i}, s_{n+i})$. If x_i is not encrypted, $(r_i, s_i) = (r_{n+i}, s_{n+i}) = (\mathbb{I}, \mathbb{I})$, where $\mathbb{I}$ is general notation for an appropriate identity mapping. If M is to be used with a compatible Turing platform, at least one output component must be in plaintext, otherwise the Turing platform will not move its finite control correctly.

2. Select the input components that are assumed to be encrypted. Unencrypted input components have $(r_i, s_i) = (\mathbb{I}, \mathbb{I})$, where $n + S < i \leq n + S + I$. Otherwise no special restrictions are necessary.

3. Select the output components to encrypt. Unencrypted output components have $(r_i, s_i) = (\mathbb{I}, \mathbb{I})$, where $S < i \leq S + O$. Otherwise no special restrictions are necessary.

4. Steps 1–3, along with the representation of H should now dictate a univariate encryption template. Generate key pairs in accordance with the univariate encryption template, and apply them to H to produce $E(H)$ using univariate encryption.

The encrypted automaton is used in precisely the same way as the unencrypted one, the only difference being that certain inputs, along with part of or all of the initial state may have to be encrypted.

An encryption template that has a plaintext output that can be used to specify the movement direction of the finite control of a Turing platform T compatible with M is called *applicable to* (M, T). An encryption template that can be applied to the representation of M is called *applicable*.

5.1.2 M Is a Register Automaton

Univariate encryption of a register automaton M proceeds as follows:

1. Select which of the register components $\vec{R}_1, \ldots, \vec{R}_m$ to encrypt. $\vec{C}$ and $\vec{D}$ must always be in plaintext if the automaton is to function properly, so the keys for their components are always set to the identity mappings.

2. The next instruction pointer mapping f has output in the form of one d-component vector, which must always be in plaintext.

3. The next storage pointer mapping g has output in the form of one d-component vector, which must always be in plaintext.

4. The mapping giving the output value $\vec{S}'$ to be written to storage cell $\vec{D}$ typically encrypts its output.

5. The register transition mapping only computes new values for some of the register components. The components are encrypted with the inverses of the keys used to decrypt the registers prior to application of the transition mapping itself. Those components reserved as passive inputs writable by some external entity have their keys set to the identity mapping.

6. Steps 1–5, along with the representation of the mappings, should now dictate a univariate encryption template. Generate key quadruples in accordance with the univariate encryption template, and apply them to the mappings to produce their encryptions.

The encrypted register automaton is used exactly as the plaintext version.

5.2 Multivariate Encryption of Programs

Section 4.5, describes how to encrypt a mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. This section describes how multivariate encryption is applied to computing endomorphisms and register automata to achieve encrypted computation. Fix an automaton M (either a Mealy machine, BSS' automaton, or register automaton) with a representation H. H is either a polynomial mapping, or a mapping defined using a function table. M may or may not be augmented and/or obfuscated.

If M is a Mealy machine, H is the mapping (δ, λ). If M is a BSS' automaton, H is the computing endomorphism given in equation 2.16. If M is a register automaton, there is no directly corresponding computing endomorphism as with Mealy machines and BSS' automata, but rather a series of evaluations of encrypted mappings.

Recall that the general form of H for Mealy and BSS' automata is:

$$H : \mathbb{Z}_N^S \times \mathbb{Z}_N^I \longrightarrow \mathbb{Z}_N^S \times \mathbb{Z}_N^O. \tag{5.2}$$

5.2.1 M Is a Mealy Or BSS' Machine

The representation H is the mapping (δ, λ). Encrypting this mapping such that it still can carry out the original computation is done as follows:

1. Select the blocks of state components that are to be stored in encrypted form. Each block stored in encrypted form must be decrypted and encrypted by the same key for each computation step.

Thus if $\vec{w}_i$, $1 \leq i \leq a$, a being the number of blocks storing all the state components, is encrypted, one must have $(r_i, s_i) = (r_{l+i}, s_{l+i})$. If $\vec{w}_i$ is not encrypted, $(r_i, s_i) = (r_{l+i}, s_{l+i}) = (\mathbb{I}, \mathbb{I})$, where $\mathbb{I}$ is general notation for an appropriate identity mapping. If M is to be used with a compatible Turing platform, at least one output block must be in plaintext, otherwise the Turing platform will not move its finite control correctly.

2. Select the blocks of input components that are assumed to be encrypted. Unencrypted blocks have $(r_i, s_i) = (\mathbb{I}, \mathbb{I})$, where $a < i \leq k - l$. Otherwise no special restrictions are necessary.

3. Select the output components to encrypt. Unencrypted output components have $(r_i, s_i) = (\mathbb{I}, \mathbb{I})$, where $a < i \leq l$. Otherwise no special restrictions are necessary.

4. Steps 1–3, along with the representation of H should now dictate a multivariate encryption template. Generate key triples in accordance with the multivariate encryption template, and apply them to H to produce $E(H)$ using multivariate encryption.

5.2.2 M Is a Register Automaton

Multivariate encryption of a register automaton M proceeds as follows:

1. Select which of the registers $\vec{R}_1, \ldots, \vec{R}_m$ to encrypt. Block length should preferably be d, such that each register has its own block. Define one variable block consisting of d components for $\vec{\mathcal{P}}_{\vec{C}}$, $\vec{S}_{\vec{D}}$, $\vec{C}$, and $\vec{D}$. $\vec{C}$ and $\vec{D}$ must always be in plaintext if the automaton is to function properly, so the keys for their blocks are always set to the identity mappings.

2. The next instruction pointer mapping f has output in the form of one d-component vector, which must always be in plaintext.

3. The next storage pointer mapping g has output in the form of one d-component vector, which must always be in plaintext.

4. The mapping giving the output value $\vec{S}'$ to be written to storage cell $\vec{D}$ typically encrypts its output.

5. The register transition mapping only computes new values for some of the blocks in the register. The mapping's component blocks are encrypted with the inverses of of the keys used to encrypt the register blocks prior to application of the transition mapping itself. Those blocks reserved as passive inputs have their keys set to the identity mapping.

6. Steps 1–5, along with the representation of the mappings, should now dictate a multivariate encryption template. Generate key quadruples in accordance with the multivariate encryption template, and apply them to the mappings to produce their encryptions.

The encrypted register automaton is used exactly as the plaintext version.

5.3 Results

It is not immediately obvious that the encrypted automata do the same computations as the plaintext versions, so some additional results are needed to establish this.

LEMMA 50 *Fix H an encryptable representation of a Mealy machine, BSS' automaton, or register automaton, an applicable encryption template U, and a set of key tuples K generated using U. For every run R of H there exists a run R' of $E(H)$ which is equivalent up to encryption under U and K.*

PROOF: Follows from the construction in definition 15_____ □.

LEMMA 51 *For every encryptable representation of a Mealy machine, BSS' automaton, or register automaton H, every univariate or multivariate encryption template U applicable to H, and every set of keys generated using U, there exists a computationally equivalent encrypted representation $E(H)$.*

PROOF: Fix any encryptable representation H, any encryption template U applicable to H, and any set of keys K generated using U. If H represents a Mealy machine or BSS' automaton, then by lemma 50, every run R of H has a corresponding run R' for $E(H)$ which is equivalent up to encryption. Since univariate and multivariate encryption are bijection with respect to non-functional data, there can only be one run R' corresponding to a unique R. The same argument is applied to every one of a register automaton's mappings, the difference being that each mapping is considered to have its own run. Since this holds for arbitrary H, U, and K, it must hold for all H, appropriate U, and K generated from U _____ □.

THEOREM 52 *For every pair (M,T), M a properly defined Mealy or BSS' machine and T a compatible Turing platform, every univariate or multivariate encryption template U applicable to (M,T), and every set of key pairs K generated using U, there exists a computationally equivalent pair $(E(M),T)$, where $E(M)$ is a Mealy or BSS' automaton, respectively.*

PROOF: Fix any (M,T), a U applicable to (M,T), and K generated using U. (M,T) and $(E(M),T)$ are computationally equivalent if they satisfy the requirements in definition 17.

REQUIREMENT 1: M has an encryptable representation H. By lemma 51, there exists a computationally equivalent $E(H)$, which is $E(M)$'s representation.

REQUIREMENT 2: Trivially satisfied, as U is applicable to (M,T).

REQUIREMENT 3: By the steps used to construct an applicable U for univariate and multivariate encryption, and the fact that M and $E(M)$ read all their input from T's tape, The keys used to decrypt input from T's tape must be the inverses of those used to encrypt output to T's tape. Furthermore all initially non-blank tape cells must be encrypted under the

same (output) encryption keys for $E(M)$'s run. Similarly, the initial state of $E(M)$ is also encrypted by these same keys. Thus $C(i,0)' = E(C(i,0))$ for all non-blank storage cells i. By the computational equivalence of H and $E(H)$, The decryption of all $C(i,a)'$ must equal the corresponding $C(i,a)$, for otherwise there would have to be some difference between two runs with initial tape contents equivalent up to encryption, which would in turn contradict the computational equivalence of H and $E(H)$. Thus one must also have $C(i,a)' = E(C(i,a))$ for all subsequent computation steps $a > 1$ $\square$.

THEOREM 53 *For every properly defined turing machine t, it is possible to generate a pair $(E(M),T)$ computationally equivalent to a pair (M,T) under a univariate or multivariate encryption template U and keys generated using U, that simulates t on T's tape, where M and $E(M)$ are both Mealy machines, BSS' automata, or register machines that only read input from the tape of a compatible Turing platform T.*

PROOF: Fix t. By theorem 1, there exists a pair (M,T) that simulates t on T's tape. Fix a univariate or multivariate encryption template U and a set of key tuples generated using U. By 52, there exists a pair $(E(M),T)$ computationally equivalent to (M,T) $\square$.

THEOREM 54 *Univariate and multivariate encryption allow cryptographically weak encrypted (Turing) universal computation.*

PROOF: Select a properly defined universal Turing machine as t. By theorem 53 there exists a pair $(E(M),T)$ which simulates t's computation on T's tape in encrypted form $\square$.

5.4 Parametric Encryption of Programs

Parametric encryption and the register automaton have been constructed for one another. Parametric encryption of a register automaton M proceeds as follows:

1. Select which of the registers $\vec{R}_1, \ldots, \vec{R}_m$ to encrypt. Block length should preferably be d, such that each register has its own block. Define one variable block consisting of d components for $\vec{\mathcal{P}}_{\vec{C}}$, $\vec{S}_{\vec{D}}$, $\vec{C}$, and $\vec{D}$. $\vec{C}$ and $\vec{D}$ must always be in plaintext if the automaton is to function properly, so the keys for their blocks are always set to the identity mappings. Program instructions may be encrypted with $\vec{C}$ as parameter.

2. The next instruction pointer mapping f has output in the form of one d-component vector, which must always be in plaintext.

3. The next storage pointer mapping g has output in the form of one d-component vector, which must always be in plaintext.

4. The mapping giving the output value $\vec{S}'$ to be written to storage cell $\vec{D}$ typically encrypts its output, and uses the block of $\vec{D}$ as parameter.

5. The register transition mapping only computes new values for some of the blocks in the register. Those blocks reserved as passive inputs have their keys set to the identity mapping. The registers that are to be encrypted may take as a parameter any other register, $\vec{C}$, $\vec{D}$, $\vec{\mathcal{P}}_{\vec{C}}$, or $\vec{S}_{\vec{D}}$.

6. Steps 1–5, along with the representation of the mappings, should now dictate a parametric encryption template. Generate key quadruples in accordance with the parametric encryption template, and apply them to the mappings to produce their encryptions.

Computations with the encrypted register automaton are done in as for the plaintext register automaton, the difference being the fact that storage cells, registers, and program instructions may be encrypted. Using the storage cell pointer as parameter for the encryption and decryption of the storage cells, effectively encrypts each cell with the same substitution cipher but with a different key for each cell. In the same way, a parametric encryption of the program instructions effectively encrypts each instruction with a different key.

Register automata encrypted with parametric encryption have the same computing power as Mealy machines and BSS' automata encrypted with univariate and multivariate encryption. Using register automata encrypted with parametric encryption in conjunction with a Turing platform, however, does not put the security provided by the parameters to good use.

Chapter 6

Conclusions

This chapter discusses the results of the previous chapters, and presents some remaining challenges.

The goal of my thesis has been to come as close as possible to providing one or more methods of encrypting automata. The encrypted automata should be able to carry out their computations as self-contained units, and use encrypted storage, while also having plaintext communication with their surroundings. Furthermore, encrypted (Turing) universal computation should also be possible.

Recall the first problem posed on the way to this goal:

PROBLEM 2 (ENCRYPTABLE REPRESENTATION) *Given a class $\mathcal{M}$ of automata, does there exist a class of representations $\mathcal{F}$ and a transformation $T_F : \mathcal{M} \longrightarrow \mathcal{F}$, such that:*

1. *elements in $\mathcal{F}$ can be used directly in computation;*
2. *T_F can be efficiently computed; and*
3. *$T_F(M)$ is encryptable?*

I have presented methods of transforming three types of finite automata into representations that satisfy all three requirements above. It turns

out that every Mealy machine and BSS' automata can be represented using one polynomial mapping, or one function table, which through iteration can execute computations. Register automata consist by definition of four different mappings that are easily encryptable such that the requirements of problem 2 are met. These mappings are termed a computing endomorphisms. It is established that the representation can be made over a ring with order $N > 1$, except for BSS' automata, where the representation must be over an $N > 1$ which is a power of one prime number.

In anticipation of the goal of universal computation, I demonstrate how these representations can be coupled to an unbounded storage in the form of an infinite series of indexed storage cells. It is established that the resulting constructions have instances capable of (Turing) universal computation, and appear to be the first such encryptable representations.

The next problem to overcome was that of the encryption itself:

PROBLEM 3 (PROGRAM ENCRYPTION) *Given a class of representations $\mathcal{F}$ satisfying property 1 in problem 2, does there exist a class of tranformations $E : K \times \mathcal{F} \longrightarrow \mathcal{F}$, where K is a class of keys, such that:*

1. *F can be encrypted in part or in its entirety;*
2. *E can be efficiently computed;*
3. *elements in K can be efficiently generated; and*
4. *$E(F)$, $F \in \mathcal{F}$, can still be used directly in computation such that:*
   (a) *one or more outputs of F may be encrypted;*
   (b) *one or more inputs of F may be encrypted; and*
   (c) *the state space/work space of F may be encrypted partially or completely?*

I have found three solutions to this problem in the form of univariate, multivariate, and parametric encryption. All three allow selective encryption of parts of the state, input, and output of the finite automata.

Parametric encryption was primarily conceived with register automata in mind, and requires special care in order to encrypt a program such that it still compute correctly.

A surprise bonus is the ability to remotely re-encrypt all automata that have been encrypted with univariate or multivariate encryption. Such re-encryption is especially interesting, as it under certain conditions gives the cryptanalyst no new information that could not have been gleaned from the original encrypted automaton itself. To my knowledge these are the first cryptosystems of *any* type with this capability.

The next step towards the goal, is making possible universal encrypted computation.

PROBLEM 4 ((TURING) UNIVERSAL ENCRYPTED COMPUTATION) *Does there exist at least one element $F \in \mathcal{F}$ such that $E(F)$ is capable of (Turing) universal encrypted computation, such that a storage of $E(F)$ bijectively mappable to a Turing machine's tape contains only encrypted values?*

This has been solved for univariate and multivariate encryption. It has also been established that all computations done in plaintext by a plaintext automaton, have a corresponding encrypted computation.

The final hurdles to achieving practical universal encrypted computation are the following two problems:

PROBLEM 5 (STRONG PROGRAM ENCRYPTION) *Does there exist a class of transformations $E : K \times \mathcal{F} \longrightarrow \mathcal{F}$ satisfying problem 3 such that:*

1. *the encrypted portion of the program representation is strongly encrypted;*
2. *the encrypted portion of the state of the encrypted program is strongly encrypted;*
3. *encrypted output is strongly encrypted; and*
4. *encrypted input is strongly encrypted?*

PROBLEM 6 (STRONG (TURING) UNIVERSAL ENCRYPTED COMPUTATION) *Does there exist a class of transformations satisfying problem 5 and an element $F \in \mathcal{F}$ such that $E(F)$ also satisfies problem 4?*

Unfortunately, the cryptanalysis of all three algorithms demonstrates quite clearly that they all have significant weaknesses. Therefore these two problems remain unsolved. This does not automatically render the ciphers useless for all practical purposes. They may still be of use in niche applications. The main problem is that the encryption of an automaton only gives that automaton the ability to apply a monoalphabetic substitution cipher with small fixed key. This opens up for a other attacks even though the value of the keys are never known.

The limiting factor for univariate and multivariate encryption is the unicity distance. This greatly limits their use in cryptographically secure encrypted computation as an iterated function system. Parametric encryption manages to reduce the impact of this problem, but not eliminate it, when used to encrypt register automata. Parametric encryption could also be employed for Mealy machine and BSS' automaton representations, but it is not clear how coherent encrypted computation could then be carried out. For this reason there are no results in this dissertation on the application of parametric encryption to Mealy machines or BSS' automata.

Noteworthy amongst the attacks, is the decomposition attack with almost linear time complexity, which works to a degree even if the attacker does not have any knowledge of whether univariate or multivariate encryption has been employed.

An interesting aspect is that all three of the encryption systems resist analytic solution methods as a means of cryptanalysis when at least one key has degree ≥ 5. Furthermore, multivariate encryption and parametric encryption are based on special cases of NP-hard problems. Whether their particular special versions have any impact on solvability and hardness is not yet clear. A side effect of the cryptanalyses are the reduced polynomial decomposition problems, similar to the more "traditional" polynomial decomposition problems. I have not yet seen any literature where such problems are discussed. They appear to have some interesting con-

sequences for function composition, but require further study, both in a purely mathematical context, as well as cryptographical context.

Problems 5 and 6 are still unresolved, although the lack of suggested systems for the latter suggest that it might not be solvable. The original intention of this work was to come as close as possible to solving these two problems, and some progress has been made, since problems 2–4 are now solved.

Almost all systems for secure or encrypted computation use exact encryption. The systems presented have their security limited by the fact that the involved mappings are dense, when expressed as polynomials, and therefore require a lot of storage. Boneh and Lipton point out in [11] that maybe systems over $\mathbb{Q}$ and similar fields/rings of infinite order may give the desired computational power and security, if such constructions are possible. Currently, this seems to be the most interesting direction, provided answers to the following informal questions are found:

1. How does one evaluate a conditional branch over a ring or field K of infinite order using a construction with "short" symbolic representation?

2. How does one find permutations of K (necessarily a permutation of an infinite set) that:

   (a) offer sufficient confusion,

   (b) have "short" symbolic representations, and

   (c) are efficiently computable?

3. The original BSS machines can be used to express computations over K. The challenge, however, is this: how does one express such a machine as an encryptable expression such as a polynomial? The adaption to finite fields in chapter 2 was done precisely because I could not find a way of expressing a BSS automaton's evaluation of the branch condition over an infinite field.

Appendix A

Notation

Special Symbols

β	The next node function of a BSS automaton
δ	The state transition function of a finite automaton
$\tilde{\delta}$	The exact polynomial interpolation of the function δ
Δ	Usually the output alphabet of a Mealy machine
λ	The output function of a Mealy machine
Σ	Usually a finite alphabet
B	The blank (tape) symbol
$E(f)$	The (possibly partial) encryption of the polynomial f
H	Usually the computing endomorphism for a finite automaton
I	Indices of the variables that are decrypted (or processed in an encrypted state)
J	Indices of the function components that are encrypted after processing
c_i	Usually a block size in multivariate or parametric encryption
g_i	Index of the parameter block in parametric encryption
r_i	Usually permutations over $\mathbb{Z}_N$
s_i	Usually the inverse of r_i
$\overline{I}$	The input space of a BSS automaton
$\overline{N}$	The set of nodes of a BSS automaton

$\overline{S}$	The state space of a BSS automaton
$\mathbb{I}$	The identity mapping
$\mathbb{O}$	An oracle—usually a host system and its environment
$\mathcal{O}(\cdot)$	"big-oh" notation for complexity measures
$\mathbb{S}$	The full state space
$\vec{S}$	Usually the value of a storage cell of a register automaton
$\mathbb{N}$	The natural numbers
$\mathcal{P}$	The set of program instructions for a register automaton
$\vec{\mathcal{P}}$	An instruction for a register automaton
$\mathbb{Q}$	The rational numbers
$\mathcal{R}$	The registers of a register automaton
$\vec{\mathcal{R}}$	A register vector of a register automaton
$\mathbb{Z}_n$	The ring of integers modulo n

Operators

$A \times B$	The Cartesian product of sets A and B
A^n	The Cartesian product of a set A with itself n times
$g \circ f$	The functional composition of g with f
$A \Rightarrow B$	A logically implies B
$A \Leftrightarrow B$	A is logically equivalent to B
$a \leftarrow b$	a is assigned the value b
$a \mod b$	The remainder of the division a/b
$a \equiv b(\mod c)$	a is equivalent to b modulo c

Indexing and Similar Things

f_i	The i^{th} component of a mapping f
x_i	The i^{th} component of a vector x
$i_{q,k}$	The k^{th} component of the vector $\vec{i}_q$
$f : A \longrightarrow B$	f maps the set A to the set B
$f\|_D$	The mapping f with domain restricted to D
$\vec{x}(n)$	The vector x after n computation steps/iterations
$\vec{x}_i$	The i^{th} of a finite ordered collection of vectors
$\vec{x}_i(n)$	The i^{th} of a finite ordered collection of vectors

	after n computation steps/iterations
$A[x, y]$	The finite extension of the group A
$\mathbb{Z}_n[x]/p(x)$	The ring of polynomials over $\mathbb{Z}_n$ modulo $p(x)$

Appendix B

Miscellaneous Proofs

B.1 A Modified Turing Machine

For notational and other practical reasons, this thesis uses primarily a modified version of the standard one-headed Turing machine with one semi-infinite tape, and two directions of movement. The version used has two directions of movement (left and right) as the standard Turing machine, but may in addition choose *not* to move its head or finite control.

The modified Turing machine (also referred to as such in the body of the thesis), is an automaton $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where Q is the set of states, Σ the input alphabet, $\Gamma = \Sigma \cup \{B\}$, B is the symbol denoting a blank cell, q_0 is the initial state, F is the set of final states. The only change actually introduced is in δ, which is now a function on the form:

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 0, 1\}, \tag{B.1}$$

where "-1" symbolizes a move to the left, "1" a move to the right, and "0" no move.

The behavior of the modified Turing machine is defined by the following mappings:

- $P : \mathbb{N} \longrightarrow \mathbb{N}$, $P(0) = 0$, which gives the position as a function of completed computation steps.

- $C : \mathbb{N} \times \mathbb{N} \longrightarrow \Gamma$, which gives the contents of the cell as a function of position and completed computation steps.
- $H : \mathbb{N} \longrightarrow Q$, $H(0) = q_0$, which gives the state as a function of completed computation steps.

The above mappings must satisfy the following:

$$\begin{array}{ll}(\forall i) & (i \in \mathbb{N} \rightarrow \\ & (P(i+1) = P(i) + \delta_3(H(i), C(P(i), i))) \wedge \\ & (C(P(i), i+1) = \delta_2(H(i), C(P(i), i))) \wedge \\ & (H(i+1) = \delta_1(H(i), C(P(i), i))))\end{array} \tag{B.2}$$

THEOREM 55 *Any Turing machine with a finite control capable of moving left, right, or standing still, may be simulated by a standard Turing machine.*

PROOF: by construction. In the following, S symbolizes the "no move" direction for the finite control. Fix a modified Turing machine $T = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. The mapping δ may be defined by a set of quintuples in $Q \times \Gamma \times Q \times \Gamma \times \{L, S, R\}$. For each quintuple on the form $(q_1, \gamma, q_2, \gamma', S)$, with $\gamma, \gamma' \in \Gamma$:

- replace it with the quintuple $(q_1, \gamma, q_2', \gamma', R)$, and
- the set of quintuples $\{(q_2', \gamma, q_2, \gamma, L) \mid \gamma \in \Gamma\}$; and
- add the state q_2' to Q.

□.

B.2 Notes on the Composition Operation

Define for N a power of a prime, a *reduction modulo N* by:

$$R(x^n) = x^{1+(n-1) \mod N}, \qquad n \in \mathbb{N} \tag{B.3}$$

The reduction operation is merely a notational convenience to clearly denote the exponent reduction that can be carried out on finite fields of the form $\mathbb{Z}_N$. This reduction gives a function equivalent to the unreduced form over $\mathbb{Z}_N$.

LEMMA 56 *Fix $N = P^e$ as a power e of a prime P. If $e = 1$, $\mathbb{Z}_N$ is interpreted as the integers modulo N. Otherwise, $\mathbb{Z}_N$ is interpreted as the set of degree $e-1$ polynomials over $\mathbb{Z}_P$ modulo an irreducible degree e polynomial $p(x)$. Given the polynomials $f(x) = \sum_{i=0}^{N-1} a_i x^i$ and $g(x) = \sum_{i=0}^{2N-2} b_i x^i$, the following holds over $\mathbb{Z}_N$:*

$$R(f(x)g(x)) = R(f(x)Rg(x)). \tag{B.4}$$

PROOF:

$$Rg(x) = b_0 + \sum_{i=1}^{N-1}(b_i + b_{N-1+i})x^i$$

So

$$\begin{aligned} R(f(x)Rg(x)) &= R\left(\left(\sum_{i=0}^{N-1} a_i x^i\right)\cdot\left(b_0 + \sum_{j=0}^{N-1}(b_j + b_{N-1+j})x^j\right)\right) \\ &= R\left(\sum_{j=0}^{N-1} b_0 a_j x^j + \sum_{i=1}^{N-1}\sum_{j=0}^{N-1}(b_i + b_{N-1+i})a_j x^{i+j}\right) \\ &= \sum_{j=0}^{N-1} b_0 a_j x^j + \sum_{i=1}^{N-1}\sum_{j=0}^{N-1}(b_i + b_{N-1+i})a_j x^{(1+(i+j-1)\bmod(N-1))} \end{aligned} \tag{B.5}$$

$$
\begin{aligned}
R(f(x)g(x)) &= R\left(\left(\sum_{i=0}^{N-1} a_i x^i\right)\cdot\left(\sum_{j=0}^{2N-2} b_j x^j\right)\right)\\
&= R\left(\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} b_i a_j x^{i+j} + \sum_{i=1}^{N-1}\sum_{j=0}^{N-1} b_{N-1+i} a_j x^{N-1+i+j}\right)\\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} b_i a_j x^{(1+(i+j-1)\bmod(N-1))}\\
&\quad + \sum_{i=1}^{N-1}\sum_{j=0}^{N-1} b_{N-1+i} a_j x^{(1+(N-1+i+j-1)\bmod(N-1))}\\
&= \sum_{j=0}^{N-1} b_0 a_j x^j + \sum_{i=1}^{N-1}\sum_{j=0}^{N-1}(b_i + b_{N-1+i}) a_j x^{(1+(i+j-1)\bmod(N-1))} \qquad \text{(B.6)}
\end{aligned}
$$

Thus $R(f(x)g(x)) = R(f(x)Rg(x))$ ——————————————— □.

LEMMA 57 *Fix $N = P^e$ as a power e of a prime P. If $e = 1$, $\mathbb{Z}_N$ is interpreted as the integers modulo N. Otherwise, $\mathbb{Z}_N$ is interpreted as the set of degree $e-1$ polynomials over $\mathbb{Z}_P$ modulo an irreducible degree e polynomial $p(x)$. Given the polynomials $f \in \mathbb{Z}_N[x_1,\ldots,x_n]$ and $g_1,\ldots,g_n \in \mathbb{Z}_N[x_1,\ldots,x_m]$, the symbolic composition of f with $g_1,\ldots,g_n$, $h \in \mathbb{Z}_N[x_1,\ldots,x_m]$, has a unique evaluation equivalent polynomial $h' \in \mathbb{Z}_N[x_1,\ldots,x_m]$ with no monomial having degree greater than $N-1$ in any one variable.*

PROOF: Symbolic functional composition of f with $g_1,\ldots,g_n$ consists of:

1. substituting the polynomials $g_1,\ldots,g_n$ for the variables $x_1,\ldots,x_n$,
2. multiplying the g_is with themselves (to exponentiate them) or with other g_js,

3. multiplying a product of g_is with a constant, and
4. adding the resulting polynomials.

By lemma 56 and the associativity of polynomial multiplication and polynomial addition, $R(f(g_1, \ldots, g_n))$ has degree $\leq N - 1$ for every variable. By corollary 1.8 in [24], the reduced polynomial is unique. Furthermore, it is the equivalent function of the unreduced version _____ □.

B.3 Keys Versus Functions for Univariate Encryption

There are $(N^n)^{(N^m)}$ possible mappings $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$. For the purposes of univariate encryption, one must have $N \geq 2$. Univariate encryption as applied f can make use of at most $n + m$ different key pairs. Each key pair can be selected in $N!$ ways. A set of $n + m$ key pairs can therefore be selected in $(N!)^{n+m}$ ways.

Define $K(N, n, m) = (N!)^{n+m}$ and $F(N, n, m) = (N^n)^{(N^m)}$.

LEMMA 58 *For all $N, n, m \geq 2$, $F(N, n, m) > K(N, n, m)$.*

PROOF: by induction on every variable. $F(2, 2, 2) = 256$ and $K(2, 2, 2) = 16$, so $F(2, 2, 2) > K(2, 2, 2)$.

- Induction on n: $F(N, n, m) > K(N, n, m) \implies F(N, n + 1, m) > K(N, n + 1, m)$.

$$\begin{array}{rcl} F(N, n+1, m) & > & K(N, n+1, m) \\ & \Updownarrow & \\ \left(N^{n+1}\right)^{(N^m)} & > & (N!)^{n+1+m} \\ & \Updownarrow & \\ \left(N^{(nN^m+N^m)}\right) & > & N!K(N, n, m) \\ & \Updownarrow & \\ N^{(N^m)}F(N, n, m) & > & N!K(N, n, m) \end{array}$$

This induction hypothesis holds if $N^{(N^m)} > N!$. Since $N \geq 2$ and $m \geq 2$, this must be the case.

- Induction on m: $F(N,n,m) > K(N,n,m) \implies F(N,n,m+1) > K(N,n,m+1)$.

$$\begin{array}{rcl} F(N,n+1,m) & > & K(N,n+1,m) \\ & \Updownarrow & \\ (N^n)^{\left(N^{m+1}\right)} & > & (N!)^{n+m+1} \\ & \Updownarrow & \\ N^{(nN^m)}N^{n((N-1)N^m)} & > & N!K(N,n,m) \\ & \Updownarrow & \\ N^{n((N-1)N^m)}F(N,n,m) & > & N!K(N,n,m) \end{array}$$

This induction hypothesis holds if $N^{n((N-1)N^m)} > N!$. Since $N \geq 2$, $n \geq 2$, and $m \geq 2$, $N^{n((N-1)N^m)} > N^{2(N^m)} > N!$, so this must be the case.

- Induction on N:

$$\begin{array}{rcl} F(N+1,n,m) & > & K(N+1,n,m) \\ & \Updownarrow & \\ ((N+1)^n)^{((N+1)^m)} & > & (N+1)!^{n+m} \\ & \Updownarrow & \\ (N+1)^{n\left(N^m+\sum_{j=0}^{m-1}\binom{m}{j}N^j\right)} & > & (N+1)^{n+m}K(N,n,m) \\ & \Updownarrow & \\ (N+1)^{n(N^m)}(N+1)^{n\left(\sum_{j=0}^{m-1}\binom{m}{j}N^j\right)} & > & (N+1)^{n+m}K(N,n,m) \end{array}$$

Since

$$(N+1)^{n(N^m)} = \left(N^{n(N^m)} + \sum_{i=0}^{nN^m-1} \binom{nN^m}{i} N^i\right)$$
$$= F(N,n,m) + \sum_{i=0}^{nN^m-1} \binom{nN^m}{i} N^i, \quad \text{(B.7)}$$

and

$$n\left(\sum_{j=0}^{m-1} \binom{m}{j} N^j\right) \geq n + n\frac{m!}{1!(m-1)!}N = n + nmN > n + m, \quad \text{(B.8)}$$

it follows that

$$(N+1)^{n(N^m)}(N+1)^{n\left(\sum_{j=0}^{m-1}\binom{m}{j}N^j\right)} >$$
$$F(N,n,m)(N+1)^{n\left(\sum_{j=0}^{m-1}\binom{m}{j}N^j\right)} \geq$$
$$F(N,n,m)(N+1)^{n+nmN} >$$
$$(N+1)^{n+m}F(N,n,m) > (N+1)^{n+m}K(N,n,m). \quad \text{(B.9)}$$

Thus all three induction hypotheses hold _____ □.

THEOREM 59 *For all $N \geq 2$, and n and m positive integers such that at least one is greater than 1: $(N^n)^{(N^m)} > (N!)^{n+m}$.*

PROOF: Define $K(N,n,m) = (N!)^{n+m}$ and $F(N,n,m) = (N^n)^{(N^m)}$. Then:

- $F(2,1,1) = 4$ and $K(2,1,1) = 4$,
- $F(2,1,2) = 16$ and $K(2,1,2) = 8$,
- $F(2,2,1) = 16$ and $K(2,2,1) = 8$.

By lemma 58, $F(N,n,m) > K(N,n,m)$ for all $N,n,m \geq 2$_____ □.

LEMMA 60 *For all* $N \geq 3$, *and* $n = m = 1$, $(N^n)^{(N^m)} < (N!)^{n+m}$.

PROOF:

$$\begin{array}{rcl} N^N & < & (N!)^2 \\ & \Updownarrow & \text{Stirling's lower bound on factorials} \\ N^N & < & 2\pi N N^{2N} e^{-2N} \leq (N!)^2 \\ & \Updownarrow & \\ 1 & < & 2\pi N N^N e^{-2N} \leq (N!)^2 N^{-N} \end{array}$$

The left inequality in the last equation above holds if $N > e^2 \approx 7.389$. If $N \geq 8$, then $F(N,1,1) < K(N,1,1)$. Compute F and K for the remaining values of N:

N	$F(N,1,1)$	$K(N,1,1)$
3	27	36
4	256	576
5	3 125	14 400
6	46 656	518 400
7	823 543	25 401 600

□.

Appendix C

Composition Using Function Tables

A side effect of this work are some simple and efficient algorithms for symbolic function composition. They can also be applied to polynomials, and would have had linear time complexity, were it not for the fact that there appears to be no known algorithm for symbolic interpolation that has linear time complexity in the number of coefficients employed. One of the faster interpolation algorithms is one by Canny et.al. [15].

Symbolic function composition may be done directly on the functions' symbolic representation in a straightforward manner or with more complicated (and efficient) approaches for special cases, such as sparse polynomials, employing tools like Fourier transforms. In some cases, however, a reasonably efficient symbolic function composition may be achieved by using a function tables.

C.1 Complexity Notation

To make the complexity results as general as possible, all operations assumed to be atomic for the purposes of this document are given their own time- and space-complexity symbols. Some of the operations are compos-

ite operations and may be implemented as algorithms with non-constant space- and/or time-complexity. Symbols are mainly divided into those for operations over $\mathbb{Z}_N$ and those over $\mathbb{Z}$. I argue that these types of complexity results, although more cumbersome, are potentially more useful in a real-world implementation situation. In such a situation, one may discover that the algorithm of choice is suddenly rendered useless by a bottleneck of the available hardware or software. An example of this is when indexing operations (such as those for nested hash-structures) equal or surpass the complexity of other operations in an algorithm implementation.

$T_{a,N}$ The time complexity of doing one integer addition modulo N when the arguments being added are elements of the field $\mathbb{Z}_N$.

$T_{m,N}$ The time complexity of doing one integer multiplication modulo N when the arguments being added are elements of the field $\mathbb{Z}_N$.

$T_{d,N}$ The time complexity of doing one integer division modulo N when the arguments being added are elements of the field $\mathbb{Z}_N$.

$T_{R,N}$ The time complexity of evaluating a binary total order relation over $\mathbb{Z}_N$.

$T_{\leftarrow,N}$ The time complexity of assigning a variable over $\mathbb{Z}_N$ a value in $\mathbb{Z}_N$.

T_a The time complexity of doing one integer addition with integers in $\mathbb{Z}$.

T_m The time complexity of doing one integer multiplication with integers in $\mathbb{Z}$.

T_d The time complexity of doing one integer division with integers in $\mathbb{Z}$.

$T_{p,n,m}$ The time complexity of computing the mth power of the integer n.

T_R The time complexity of evaluating a binary total order relation over $\mathbb{Z}$.

T_{and} The time complexity of logical and.

T_{or} The time complexity of logical or.

$T_{\leftarrow}$ The time complexity of assigning a variable over $\mathbb{Z}$ a value in $\mathbb{Z}$.

T_i Time complexity of general index referencing overhead for non-nested indices.

$S_{a,N}$ The space complexity of doing one integer addition modulo N when the arguments being added are elements of the field $\mathbb{Z}_N$.

$S_{m,N}$ The space complexity of doing one integer multiplication modulo N when the arguments being added are elements of the field $\mathbb{Z}_N$.

$S_{d,N}$ The space complexity of doing one integer division modulo N when the arguments being added are elements of the field $\mathbb{Z}_N$.

$S_{s,N}$ The space complexity of storing an integer in $\mathbb{Z}_N$.

S_a The space complexity of doing one integer addition with arguments in $\mathbb{Z}$.

S_m The space complexity of doing one integer multiplication with arguments in $\mathbb{Z}$.

S_d The space complexity of doing one integer division with arguments in $\mathbb{Z}$.

$S_{p,n,m}$ The space complexity of computing the mth power of the integer n.

S_s The space complexity of storing an integer in $\mathbb{Z}$.

The following assumptions

ASSUMPTION 1 *Every arithmetic operations is assumed to have time complexity that is a function of the size of its input measured in bits.*

ASSUMPTION 2 *Every integer n occupies $\lceil \log_2(\max\{|n|\}+1) \rceil$ storage units, where $\max\{|n|\}$ is the maximal absolute value taken by n during the execution of the algorithm.*

A more precise computation of the storage occupied by every integer is $\lceil \log_2(\max\{|n|\} + 1) \rceil + 1$, which takes into account the sign of n. It is not, however, possible to generally assume that n has a sign, as this is a representational issue, which may vary from implementation to implementation. Another issue is the storage of the length of the integer itself. This is necessary if more storage may vary from integer to integer. This also, though is very implementation specific, and the length does not necessarily have to be recorded individually with each integer. So this is also excluded from the computation of occupied storage.

ASSUMPTION 3 *Evaluation of logical expressions does not use short-circuiting to increase efficiency.*

ASSUMPTION 4 *For a fixed N, all atomic operations modulo N are assumed to have constant space and time complexity.*

ASSUMPTION 5 *Subtraction has the same space and time complexity as addition.*

ASSUMPTION 6 *Interpolation polynomials almost always have dense representations.*

ASSUMPTION 7 *Indexing operations are significant operations with respect to complexity.*

ASSUMPTION 8 *Indexes are integers in $\mathbb{Z}$.*

ASSUMPTION 9 *Index addition and multiplication is always taken to have time complexity T_a and T_m, respectively, regardless of whether it is done modulo N or not.*

ASSUMPTION 10 *Index addition and multiplication is always taken to have space complexity S_a and S_m, respectively, regardless of whether it is done modulo N or not.*

ASSUMPTION 11 *Indexes represented using vectors over $\mathbb{Z}_N$, have operations on them carried out on each individual component. The space and time complexity this entails is additive for each component.*

ASSUMPTION 12 *A table lookup in an ordered table has average time complexity T_o.*

ASSUMPTION 13 *A table lookup in an unordered table has average time complexity T_u.*

ASSUMPTION 14 *A table lookup in an ordered table has average space complexity S_o.*

ASSUMPTION 15 *A table lookup in an unordered table has average space complexity S_u.*

ASSUMPTION 16 *All subalgorithms are assumed to be inline code (with appropriate renaming of variables) for the purposes of computing complexity.*

As a convention, storage complexity includes local algorithm workspace and the space used to store input, but does not include the space for the implementation of the algorithm itself.

C.2 Vectorized Indices

Before beginning on the main composition algorithms themselves, some of the operations employed, should be reviewed. Since the indexes and mappings to be encrypted have no inherent limitation on the number of components they are allowed, nested loop constructions will not have sufficient generality. This means that all vectorized indexes must be handled by single, unnested loops. The following three operations, given in the form of algorithms, make this possible.

The first algorithm converts a vector of integers to one integer by considering it a number with base-N representation.

Algorithm 1

In: The vector $(x_1, \ldots, x_m) \in \mathbb{Z}_N$, integer $m \geq 1$, and integer $N > 1$.

Out: The integer $X \geq 0$.

$i \leftarrow m$, $X \leftarrow x_m$	$2T_{\leftarrow} + T_i$
while($i > 1$) do:	T_R
$\quad i \leftarrow i - 1$	$T_{\leftarrow} + T_a$
$\quad X \leftarrow NX + x_i$	$T_{\leftarrow} + T_m + T_a + T_i$

LEMMA 61 *Algorithm 1 has space complexity* $2m\lceil \log_2 N \rceil + 2\lceil \log_2(m+1) \rceil + \lceil \log_2(N+1) \rceil$*, which is* $\mathcal{O}(m \log_2 N)$.

PROOF: Storing the input requires space: $mS_{s,N} + 2S_s$. Local variables occupy a further $2S_s$. $S_{s,N} = \lceil \log_2 N \rceil$. Since $0 \leq i \leq m$ and $0 \leq X \leq N^m - 1$, these integers occupy $\lceil \log_2(m+1) \rceil$ and $m\lceil \log_2 N \rceil$ storage units, respectively. N occupies $\lceil \log_2(N+1) \rceil$ storage units. Total space complexity is therefore $m\lceil \log_2 N \rceil + 2\lceil \log_2(m+1) \rceil + \lceil \log_2(N+1) \rceil + m\lceil \log_2 N \rceil$, which is asymptotically proportional to $m \log_2 N + \log_2 m$ ——— □.

LEMMA 62 *Algorithm 1 has time complexity* $mT_R + 2mT_{\leftarrow} + (m-1)(2T_a + T_m) + mT_i$*, which is* $\mathcal{O}(m)$.

PROOF: The main loop is run $m - 1$ times in all, but the loop condition is evaluated m times, so time complexity is $2T_{\leftarrow} + T_i + mT_R + (m-1)(2T_{\leftarrow} + 2T_a + T_m + T_i) = mT_R + 2mT_{\leftarrow} + (m-1)(2T_a + T_m) + mT_i$, which is asymptotically proportional to m ——— □.

The next algorithm is the inverse of algorithm 1.

Algorithm 2

In: The integer $X \geq 0$, the integer m, and the integer $N > 1$.

Out: The vector $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$.

$i \leftarrow m$, $k \leftarrow N^{m-1}$	$2T_{\leftarrow} + T_a + T_{p,n,m-1}$
while($i > 1$) do:	T_R
$x_i \leftarrow \lfloor X/k \rfloor$	$T_i + T_{\leftarrow} + T_d$
$X \leftarrow X - x_i k$	$T_{\leftarrow} + T_a + T_i + T_m$
$i \leftarrow i - 1$	$T_{\leftarrow} + T_a$
$k \leftarrow k/N$	$T_{\leftarrow} + T_d$
$x_i \leftarrow X$	$T_i + T_{\leftarrow}$

LEMMA 63 *Algorithm 2 has space complexity* $(3m+1)\lceil \log_2 N \rceil + 2\lceil \log_2 m \rceil + S_{p,N,m-1}$*, which is* $\mathcal{O}(m \log_2 N) + S_{p,N,m-1}$.

PROOF: The integers X, m, N, i, and k have maximal sizes of $N^m - 1$, m, N, m, and N^{m-1}, respectively. The exponentiation operation requires a work space of $S_{p,N,m-1}$. The vector $(x_1, \ldots, x_m)$ occupies $mS_{s,N} = m\lceil \log_2 N \rceil$ storage units. Total space complexity is thus $3m\lceil \log_2 N \rceil + 2\lceil \log_2(m+1) \rceil + \lceil \log_2 N \rceil + S_{p,N,m-1}$ $\square$.

LEMMA 64 *Algorithm 2 has time complexity* $mT_R + (m-1)(T_R + 2T_i + 4T_{\leftarrow} + 2T_a + 2T_d + T_m) + T_i + T_a + 3T_{\leftarrow} + T_{p,N,m-1}$*, which is* $\mathcal{O}(m) + T_{p,N,m-1}$.

PROOF: Initialization contributes $2T_{\leftarrow} + T_a + T_{p,N,m-1}$. The loop is always run $m-1$ times, which contributes $(m-1)(T_R + 2T_i + 4T_{\leftarrow} + 2T_a + 2T_d + T_m)$. The loop is exited on the final evaluation of the loop condition, which contributes T_R time units. The final assignment contributes $T_i + T_{\leftarrow}$, which amounts to a total of $mT_R + (m-1)(T_R + 2T_i + 4T_{\leftarrow} + 2T_a + 2T_d + T_m) + T_i + T_a + 3T_{\leftarrow} + T_{p,N,m-1}$ $\square$.

Algorithm 3

In: The integer $N^m > X \geq 0$, the integer m, and the precomputed powers $k_i = N^{i-1}$ for $1 < i \leq m$.

Out: The vector $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$.

$i \leftarrow m$	$T_{\leftarrow}$
while($i > 1$) do:	T_R

$x_i \leftarrow \lfloor X/k_i \rfloor$	$2T_i + T_{\leftarrow} + T_d$
$X \leftarrow X - x_i k_i$	$2T_i + T_{\leftarrow} + T_a + T_m$
$i \leftarrow i - 1$	$T_{\leftarrow} + T_a$
$x_i \leftarrow X$	$T_i + T_{\leftarrow}$

Algorithm 3 trades some space complexity for the same asymptotic time complexity as algorithm 2, but with a lower constant factor.

LEMMA 65 *Algorithm 3 has space complexity*

$$\lceil \log_2(N^m) \rceil + \frac{m(m+1)}{2} \lceil \log_2 N \rceil + 2 \lceil \log_2(m+1) \rceil,$$

which is $\mathcal{O}(m^2 \log_2 N)$.

PROOF: The integers X, m, $k_1, \ldots, k_m$, and i occupy $\lceil \log_2(N^m) \rceil$, $\lceil \log_2(m+1) \rceil$, $\lceil \log_2 N \rceil, \ldots, \lceil m \log_2 N \rceil$, and $\lceil \log_2(m+1) \rceil$ storage units, respectively. The vector $(x_1, \ldots, x_m)$ occupies $m \lceil \log_2 N \rceil$ storage units. Total storage occupied is:

$$\lceil \log_2(N^m) \rceil + \frac{m(m+1)}{2} \lceil \log_2 N \rceil + 2 \lceil \log_2(m+1) \rceil,$$

which is $\mathcal{O}(m^2 \log_2 N)$ □.

LEMMA 66 *Algorithm 3 has time complexity* $mT_R + T_i + 2T_{\leftarrow} + (m-1)(4T_i + 3T_{\leftarrow} + 2T_a + T_d + T_m)$*, which is* $\mathcal{O}(m)$.

PROOF: Initialization takes $T_{\leftarrow}$ time units. The main loop is executed $m-1$ times, contributing $(m-1)(T_R + 4T_i + 3T_{\leftarrow} + 2T_a + T_d + T_m)$ time units. The last evaluation of the loop condition contributes T_R, and final assignment $T_i + T_{\leftarrow}$. Asymptotic time complexity is $\mathcal{O}(m)$ □.

Lastly, I present two algorithms for incrementing vectorized indices. The first covers vectorized indices where the base is uniform: all components are from the same set. The second covers vectorized indices where the components are not necessarily elements from the same set.

Algorithm 4

In: The vector $(v_1, \ldots, v_m) \in \mathbb{Z}_N^m$, the integers N and m.

Out: The vector $(v_1, \ldots, v_m) \in \mathbb{Z}_N^m$ incremented by $1 \mod N^m$ when viewed as an m-digit number in base N.

$i \leftarrow 1$	$T_{\leftarrow}$
do:	
$v_i \leftarrow (v_i + 1) \mod N$	$2T_i + T_{\leftarrow} + T_{a,N}$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while ($v_{i-1} = 0$ and $i \leq m$)	$T_a + T_i + 2T_R + T_{\text{and}}$

LEMMA 67 *Algorithm 4 has space complexity* $m\lceil \log_2 N \rceil + 2\lceil \log_2(m+1) \rceil + \lceil \log_2(N+1) \rceil$ *which is* $\mathcal{O}(m \log_2 N)$.

PROOF: The vector $(v_1, \ldots, v_m)$ occupies $m\lceil \log_2 N \rceil$ storage units. The integers i, m, and N occupy in all $2\lceil \log_2(m+1) \rceil + \lceil \log_2(N+1) \rceil$ storage units. The total is $m\lceil \log_2 N \rceil + 2\lceil \log_2(m+1) \rceil + \lceil \log_2(N+1) \rceil$ which is $\mathcal{O}(m \log_2 N)$ $\square$.

LEMMA 68 *Algorithm 4 has minimum time complexity* $3T_i + T_{a,N} + 2T_a + 3T_{\leftarrow} + 2T_R + T_{\text{and}}$, *which is* $\mathcal{O}(1)$.

PROOF: Initialization takes $T_{\leftarrow}$ time units. The loop is always run at least once, and thus contributes $3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}}$ $\square$.

LEMMA 69 *Algorithm 4 has maximum time complexity* $T_{\leftarrow} + m(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}})$, *which is* $\mathcal{O}(m)$.

PROOF: Initialization takes $T_{\leftarrow}$ time units. The loop is run a maximum of m times, contributing $m(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}})$ $\square$.

LEMMA 70 *Under the assumption that all* $(v_1, \ldots, v_m) \in \mathbb{Z}_N^m$ *are equally likely as input (their distribution is uniform), algorithm 4 has average time complexity*

$$T_{\leftarrow} + \frac{(N^m - 1)}{(N^m - N^{m-1})}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}}),$$

which is $\mathcal{O}((N^m - 1)/(N^m - N^{m-1}))$.

PROOF: Fix N and m. Initialization takes $T_{\leftarrow}$ time units. The average time complexity of the loop for uniformily distributed input can be computed by dividing the total number of loop runs by N^m, the number of possible inputs. The loop is always executed at least once, so the sum is at least N^m. There is one value of the first digit which always leads to a second run through the loop, giving N^{m-1} additional runs, and so on. There are thus $\sum_{i=1}^{m} N^i = (N^{m+1} - 1)/(N - 1) - 1 = N(N^m - 1)/(N - 1)$ runs in all, giving an average of $(N^{m+1} - N)/(N^m(N - 1)) = (N^m - 1)/(N^m - N^{m-1})$ runs. This contributes an average of

$$\frac{(N^m - 1)}{(N^m - N^{m-1})}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}})$$

time units to the total_____ □.

Algorithm 5

In: The base $(c_1, \ldots, c_m)$, such that all c_i are positive integers greater than 1, the vector $(v_1, \ldots, v_m) \in \mathbb{Z}_{c_1} \times \cdots \times \mathbb{Z}_{c_m}$, the integer m.

Out: The vector $(v_1, \ldots, v_m)$ incremented by $1 \mod \prod_{i=1}^{m} c_i$ when viewed as a m-digit number in base $(c_1, \ldots, c_m)$.

$i \leftarrow 1$	$T_{\leftarrow}$
do:	
$v_i \leftarrow (v_i + 1) \mod c_i$	$3T_i + T_{\leftarrow} + T_{a,c_i}$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while ($v_{i-1} = 0$ and $i \leq m$)	$T_a + T_i + 2T_R + T_{\text{and}}$

LEMMA 71 *Algorithm 5 has space complexity*

$$2\sum_{i=1}^{m} \lceil \log_2(c_i + 1) \rceil + 2\lceil \log_2(m + 1) \rceil,$$

which is $\mathcal{O}(\sum_{i=1}^{m} \log_2 c_i)$.

PROOF: The vector $(v_1, \ldots, v_m)$ occupies $\sum_{i=1}^{m} \lceil \log_2(c_i + 1) \rceil$ storage units. So does the base $(c_1, \ldots, c_m)$. The integers i and m both occupy $\lceil \log_2(m + 1) \rceil$ storage units. Since there are m components to $\vec{v}$ and $\vec{c}$, $\log_2 m$ is asymptotically insignificant $\square$.

LEMMA 72 *Algorithm 5 has minimum time complexity* $T_{\leftarrow} + (4T_i + 2T_a + 2T_{\leftarrow} + T_R + T_{\text{and}}) + T_{a,c_1}$, *which is* $\mathcal{O}(1)$.

PROOF: Initialization contributes $T_{\leftarrow}$ time units. The single run of the main loop contributes $4T_i + 2T_{\leftarrow} + 2T_a + T_{a,c_1} + 2T_R + T_{\text{and}}$ $\square$.

LEMMA 73 *Algorithm 5 has maximum time complexity* $T_{\leftarrow} + m(4T_i + 2T_a + 2T_{\leftarrow} + T_R + T_{\text{and}}) + \sum_{i=1}^{m} T_{a,c_i}$, *which is* $\mathcal{O}(m)$.

PROOF: Initialization contributes $T_{\leftarrow}$ time units. The m runs of the main loop contribute in all $m(4T_i + 2T_a + 2T_{\leftarrow} + T_R + T_{\text{and}}) + \sum_{i=1}^{m} T_{a,c_i}$ time units $\square$.

LEMMA 74 *Under the assumption that all* $(v_1, \ldots, v_m) \in \prod_{i=1}^{m} \mathbb{Z}_{c_i}$ *are equally likely as input (their distribution is uniform), algorithm 5 has average time complexity*

$$T_{\leftarrow} + (4T_i + 2T_a + 2T_{\leftarrow} + T_R + T_{\text{and}}) \left(1 + \sum_{i=2}^{m} \prod_{j=1}^{i-1} c_j^{-1} \right) + T_{a,c_1} + \sum_{i=2}^{m} \left(T_{a,c_i} \prod_{j=1}^{m} c_j^{-1} \right),$$

which is $\mathcal{O}(1 + \sum_{i=2}^{m} \prod_{j=1}^{i-1} c_j^{-1})$.

PROOF: Fix $(c_1, \ldots, c_m)$ and m. Initialization takes $T_{\leftarrow}$ time units. The average time complexity of the loop for uniformily distributed input can be computed by dividing the total number of loop runs by $\prod_{i=1}^{m} c_i$, the number of possible inputs. There is one value of the first digit which always

leads to a second run through the loop, giving $\prod_{i=2}^{m} c_i$ additional runs, and so on. There are thus $\sum_{i=1}^{m} \prod_{j=i}^{m} c_j$ runs in all, giving an average of $1 + \sum_{i=2}^{m} \prod_{j=1}^{i-1} c_j^{-1}$ runs. This contributes an average of

$$(4T_i + 2T_a + 2T_{\leftarrow} + T_R + T_{\text{and}}) \left(1 + \sum_{i=2}^{m} \prod_{j=1}^{i-1} c_j^{-1} \right)$$

times units to the total. In addition, since T_{a,c_i} varies with the loop index, it contributes an average of $T_{a,c_1} + \sum_{i=2}^{m} \left(T_{a,c_i} \prod_{j=1}^{m} c_j^{-1} \right)$ time units___ □.

C.3 Converting Function Tables

Some of the composition algorithms depend on a transformation from multivariate mappings to isomorphic univariate mappings. A mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ may be transformed to a univariate mapping $F : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^n}$ with the following algorithm:

Algorithm 6

In: The function $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1, \ldots, f_n)$ indexed by tuples $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$, the integers N, m, and n.

Out: The function $F : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^n}$ represented as a table containing all N^m mapping values $F(X)$ for all integers $X \in \mathbb{Z}_{N^m}$.

$(x_1, \ldots, x_m) \leftarrow (0, \ldots, 0)$	$m(T_i + T_{\leftarrow})$
$X \leftarrow 0, k \leftarrow N^m - 1$	$2T_{\leftarrow} + T_{p,N,m} + T_a$
do:	
lookup $f(x_1, \ldots, x_m) = (f_1, \ldots, f_n)$	$mT_i + T_o$
$F \leftarrow N^{n-1} f_n + \cdots + N^1 f_2 + f_1$ using algorithm 1	
increment $(x_1, \ldots, x_m)$ using algorithm 4	
$F(X) \leftarrow F$	$T_i + T_{\leftarrow}$
$X \leftarrow X + 1$	$T_{\leftarrow} + T_a$
while($X \leq k$)	T_R

LEMMA 75 *Algorithm 6 has space complexity approximately* $(2nN^m + 3m + n + 1)\lceil \log_2 N \rceil + 2\lceil \log_2(m+1) \rceil + 2\lceil \log_2(n+1) \rceil + n\lceil \log_2 N \rceil$*, which is* $\mathcal{O}(nN^m \lceil \log_2 N \rceil)$.

PROOF: The original table for $f()$ occupies $nN^m S_{s,N} = nN^m \lceil \log_2 N \rceil$ storage units. Since every $F(X)$ lies between 0 and N^n, the new table for $F()$ occupies $N^m \lceil \log_2 N^n \rceil$ storage units. The integers N, m, n, k, X, and F occupy $\lceil \log_2(N+1) \rceil$, $\lceil \log_2(m+1) \rceil$, $\lceil \log_2(n+1) \rceil$, $\lceil \log_2(N^m) \rceil$, $\lceil \log_2(N^m+1) \rceil$, and $\lceil \log_2(N^n) \rceil$ storage units, respectively. The vector $(x_1, \ldots, x_m)$ occupies $m\lceil \log_2 N \rceil$ storage units. The result of the table lookup occupies $n\lceil \log_2 N \rceil$ storage units. Algorithm 1 only needs an additional integer between 1 and n for its local indexing, contributing $\lceil \log_2(n+1) \rceil$ storage units. Algorithm 4 also needs only an additional integer for its local indexing, contributing $\lceil \log_2(m+1) \rceil$ storage units. □.

LEMMA 76 *Algorithm 6 has time complexity*

$$\begin{aligned} & m(T_i + T_{\leftarrow}) + 2T_{\leftarrow} + T_{p,N,m} + T_a \\ & + N^m((m+n+1)T_i + T_o + (n+1)T_R + (2n+3)T_{\leftarrow} + (2n-1)T_a + 2(n-1)T_m) \\ & \qquad + \frac{N(N^m-1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\mathsf{and}}), \end{aligned}$$

which is $\mathcal{O}(N^m(m+n))$.

PROOF: Initialization takes $m(T_i + T_{\leftarrow}) + 2T_{\leftarrow} + T_{p,N,m} + T_a$ time units. The main loop is run N^m times. This means that algorithm 4 is used once for each of its possible N^m cases. The main loop thus contributes:

$$\begin{aligned} & N^m(mT_i + T_o + nT_R + 2nT_{\leftarrow} + (n-1)(2T_a + T_m) + nT_i + T_{\leftarrow} \\ & + T_i + 2T_{\leftarrow} + T_a + T_R) + \frac{N(N^m-1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\mathsf{and}}), \end{aligned} \tag{C.1}$$

which can also be written:

$$N^m((m+n+1)T_i+T_o+(n+1)T_R+(2n+3)T_{\leftarrow}+(2n-1)T_a+2(n-1)T_m) + \frac{N(N^m-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\text{and}}). \quad \text{(C.2)}$$

□.

The inverse of algorithm 6 is also needed.

Algorithm 7

In: The function $F : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^n}$ represented as a table containing all N^m mapping values F indexed by $X \in \mathbb{Z}_{N^m}$. The integers N, n, m, and the precomputed powers $k_i = N^{i-1}$ for $1 \leq i \leq m$.

Out: The function $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1, \ldots, f_n)$ indexed by tuples $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$

$(x_1, \ldots, x_m) \leftarrow (0, \ldots, 0)$	$mT_i + mT_{\leftarrow}$
$X \leftarrow 0, k \leftarrow N^m - 1$	$2T_{\leftarrow} + T_{p,N,m} + T_a$
do:	
$F = F(X)$	$T_{\leftarrow} + T_i$
convert F to $(f_1, \ldots, f_n) \in \mathbb{Z}_N^n$ using algorithm 3	
$f(x_1, \ldots, x_m) \leftarrow (f_1, \ldots, f_n)$	$T_o + mT_i + nT_i + nT_{\leftarrow}$
increment $(x_1, \ldots, x_m)$ using algorithm 4	
$X \leftarrow X + 1$	$T_{\leftarrow} + T_a$
while($X \leq k$)	T_R

LEMMA 77 *Algorithm 7 has a space complexity of approximately*

$$(3m + n + 2nN^m + \frac{m(m+1)}{2})\lceil \log_2 N \rceil + \lceil \log_2 m + 1 \rceil,$$

which is $\mathcal{O}(nN^m)$.

PROOF: $\vec{x}$ occupies $m\lceil \log_2 N \rceil$ storage units. $(f_1, \ldots, f_n)$ occupies $n\lceil \log_2 N \rceil$ storage units. The original table for $f()$ occupies $N^m n\lceil \log_2 N \rceil$ storage

units, while the table for $F()$ occupies $N^m\lceil\log_2(N^n)\rceil$ storage units. The precomputed powers $k_1,\ldots,k_m$ occupy $\lceil\log_2 N\rceil,\ldots,\lceil\log_2(N^m)\rceil$ storage units. X and k occupy $\lceil\log_2(N^m)\rceil$ storage units each. In addition to the variables defined explicitly in algorithm refvectorize, algorithms 3 and 4 need one additional integer, which occupies $\lceil\log_2(m+1)\rceil$ storage units □.

LEMMA 78 *Algorithm 7 has time complexity*

$$\begin{aligned} &mT_i + (m+2)T_{\leftarrow} + T_{p,N,m} + T_a + N^m((4n+2)T_{\leftarrow} \\ &\quad + (5n+m-3)T_i + (n+1)T_R + T_o + (2n-1)T_a + (n-1)(T_d+T_m) \\ &\qquad + \frac{N(N^m-1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\mathsf{and}}) \end{aligned}$$

which is $\mathcal{O}(nN^m)$.

PROOF: Initialization takes $mT_i+(m+2)T_{\leftarrow}+T_{p,N,m}+T_a$. Since algorithm 4 is run N^m times in all, and uses every possible case once, the main loop contributes:

$$\begin{aligned} N^m(&T_{\leftarrow} + T_i + nT_R + T_i + 2T_{\leftarrow} + (n-1)(4T_i + 3T_{\leftarrow} + 2T_a + T_d + T_m) \\ &+ T_o + (m+n)T_i + nT_{\leftarrow} + T_{\leftarrow} + T_a + T_R + T_{\leftarrow}) \\ &+ \frac{N(N^m-1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\mathsf{and}}) \end{aligned}$$

□.

C.4 Composition Using Function Tables

It is now possible to describe function composition using function tables.

Algorithm 8

In: The mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1,\ldots,f_n)$ indexed by tuples $(x_1,\ldots,x_m) \in$

$\mathbb{Z}_N^m$. The mapping $g : \mathbb{Z}_N^n \longrightarrow \mathbb{Z}_N^o$ represented as a table containing all N^n mapping values $(g_1, \ldots, g_o)$ indexed by tuples $(x_1, \ldots, x_n) \in \mathbb{Z}_N^n$. The integers N, m, n, and o.

Out: The composition of f and g in the form of a function $H : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^o}$ represented as a table containing all N^m mapping values $F(X) \in \mathbb{Z}_{N^o}$ for all integers $X \in \mathbb{Z}_{N^m}$.

Convert f to $F : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^n}$ using algorithm 6	
Convert g to $G : \mathbb{Z}_{N^n} \longrightarrow \mathbb{Z}_{N^o}$ using algorithm 6	
$X \leftarrow 0, k \leftarrow N^m - 1$	$2T_{\leftarrow} + T_{p,N,m} + T_a$
do:	
$H(X) \leftarrow G(F(X))$	$3T_i + T_{\leftarrow}$
$X \leftarrow X + 1$	$T_{\leftarrow} + T_a$
while($X \leq k$)	T_R

LEMMA 79 *Algorithm 8 has a space complexity of approximately*

$$(2nN^m + 2oN^n + 2m + \max\{m, n\} + \max\{n, o\})\lceil \log_2 N \rceil + \lceil \log_2(\max\{m + 1, n + 1, o + 1\}) \rceil,$$

which is $\mathcal{O}(nN^m + oN^n)$.

PROOF: The original tables for $f()$ and $g()$ occupy in all $nN^m\lceil \log_2 N \rceil + oN^n\lceil \log_2 N \rceil$ storage units. The tables for $F()$ and $G()$ occupy $N^m\lceil \log_2(N^n) \rceil + N^n\lceil \log_2(N^o) \rceil$ storage units. X and k occupy $\max\{m, n\}\lceil \log_2 N \rceil$ storage units each. Algorithm 6 requires for both its runs $\max\{m, n\}\lceil \log_2 N \rceil + \max\{n, o\}\lceil \log_2 N \rceil$ storage units for the vectors, and $\lceil \log_2(\max\{m + 1, n + 1, o + 1\}) \rceil$ storage units for the indexes required by algorithms 1 and 4. □.

LEMMA 80 *Algorithm 8 has time complexity*

$$
\begin{aligned}
&2T_{\leftarrow} + 2T_{p,N,m} + T_{p,N,n} + 3T_a + (m+n)(T_i + T_{\leftarrow}) + 4T_{\leftarrow} + \\
&+ N^m((m+n+4)T_i + T_o + (n+2)T_R + (2n+5)T_{\leftarrow} + 2nT_a + 2(n-1)T_m) \\
&+ N^n((n+o+1)T_i + T_o + (o+1)T_R + (2o+3)T_{\leftarrow} + (2o-1)T_a + 2(o-1)T_n) \\
&\qquad + \frac{N(N^m + N^n - 2)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\mathsf{and}}),
\end{aligned}
$$

which is $\mathcal{O}((m+n)N^m + (n+o)N^n)$.

PROOF: Initialization after the first two conversions takes $2T_{\leftarrow} + T_{p,N,m} + T_a$ time units. The main loop is run N^m times, consuming $N^m(3T_i + 2T_{\leftarrow} + T_a + T_R)$ time units. The two applications of algorithm 6 consume:

$$
\begin{aligned}
&m(T_i + T_{\leftarrow}) + 2T_{\leftarrow} + T_{p,N,m} + T_a \\
&+ N^m((m+n+1)T_i + T_o + (n+1)T_R + (2n+3)T_{\leftarrow} + (2n-1)T_a + 2(n-1)T_m) \\
&\qquad + \frac{N(N^m - 1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}}) \\
&\qquad\qquad + n(T_i + T_{\leftarrow}) + 2T_{\leftarrow} + T_{p,N,n} + T_a \\
&+ N^n((n+o+1)T_i + T_o + (o+1)T_R + (2o+3)T_{\leftarrow} + (2o-1)T_a + 2(o-1)T_n) \\
&\qquad + \frac{N(N^n - 1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}})
\end{aligned}
$$

time units. □.

The result of algorithm 8 may be converted into a form vectorized over $\mathbb{Z}_N$. This form may in turn be transformed into polynomial form, provided N is a power of a prime.

Algorithm 9

In: The mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1, \ldots, f_n)$ indexed by tuples $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$. The mappings $h_1, \ldots, h_k : \mathbb{Z}_N^{c_i} \longrightarrow \mathbb{Z}_N^{c_i}$, represented as tables containing all N^{c_i} mapping values respectively. The integers $N, k, m, n, c_1, \ldots, c_k$, such that $\sum_{i=1}^{k} c_i = m$.

Out: The composition of $h_1, \ldots, h_k$ and f defined as:

$$f(h_1(x_1, \ldots, x_{c_1}), h_2(x_{c_1+1}, \ldots, x_{c_1+c_2}), \ldots, h_k(x_{m-c_k+1}, \ldots, x_m)),$$

and expressed as the function $L : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^n}$.

Convert f to F using algorithm 6	
$i \leftarrow 1, K \leftarrow N^m - 1$	$2T_{\leftarrow} + T_{p,N,m} + T_a$
do:	
Convert h_i to $H_i : \mathbb{Z}_{N^{c_i}} \longrightarrow \mathbb{Z}_{N^{c_i}}$ using algorithm 6	
$y_i \leftarrow N^{c_i}$	$2T_i + T_{\leftarrow} + T_{p,N,c_i}$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while($i \leq k$)	T_R
$i \leftarrow 0$	$T_{\leftarrow}$
$(b_1, \ldots, b_k) \leftarrow (0, \ldots, 0)$	$k(T_i + T_{\leftarrow})$
do:	
$u \leftarrow 0, j \leftarrow k$	$2T_{\leftarrow}$
do:	
$u \leftarrow y_j u + H_j(b_j)$	$T_{\leftarrow} + 4T_i + T_m + T_a$
$j \leftarrow j - 1$	$T_{\leftarrow} + T_a$
while($j > 0$)	T_R
$L(i) \leftarrow F(u)$	$2T_i + T_{\leftarrow}$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
Increment $\vec{b}$ using algorithm 5	
while($i \leq K$)	T_R

LEMMA 81 *Algorithm 9 has space complexity of approximately*

$$\left(3nN^m + 3m + 4\sum_{j=1}^{k}(c_j N^{c_j}) + 1\right)\lceil \log_2 N \rceil$$
$$+ \sum_{j=1}^{k} \lceil \log_2 c_j \rceil + \lceil \log_2 k \rceil + \lceil \log_2 m \rceil + \lceil \log_2 n \rceil + \lceil \log_2(\max\{k, m\}) \rceil,$$

which is $\mathcal{O}(nN^m)$.

PROOF: The tables for the mappings $f, h_1, \ldots, h_k$ occupy

$$nN^m\lceil \log_2 N\rceil, c_1N^{c_1}\lceil \log_2 N\rceil, \ldots, c_kN^{c_k}\lceil \log_2 N\rceil$$

storage units, respectively. The tables for $F, H_1, \ldots, H_k, L$ occupy

$$N^m\lceil \log_2(N^n)\rceil, N^{c_1}\lceil \log_2(N^{c_1})\rceil, \ldots, N^{c_k}\lceil \log_2(N^{c_k})\rceil, N^m\lceil \log_2(N^n)\rceil$$

storage units, respectively. The integers $N, k, m, n, c_1, \ldots, c_k, i, j, K, u$ occupy

$$\lceil \log_2(N+1)\rceil, \lceil \log_2(k+1)\rceil, \lceil \log_2(m+1)\rceil, \lceil \log_2(n+1)\rceil, \lceil \log_2(c_1+1)\rceil, \ldots,$$
$$\lceil \log_2(c_k+1)\rceil, \lceil \log_2(N^m)\rceil, \lceil \log_2(\max\{k,m\}+1)\rceil, \lceil \log_2(N^m)\rceil, \lceil \log_2(N^m)\rceil$$

storage units, respectively. The integers $b_1, \ldots, b_k, y_1, \ldots, y_k$ occupy

$$\lceil \log_2(N^{c_1})\rceil, \ldots, \lceil \log_2(N^{c_k})\rceil, \lceil \log_2(N^{c_1}+1)\rceil, \ldots, \lceil \log_2(N^{c_k}+1)\rceil$$

storage units, respectively. Algorithm 6 needs an additional vector $(x_1, \ldots, x_m)$, which occupies $m\lceil \log_2 N\rceil$ storage units. Algorithm 5 doesn't require any additional variables $\square$.

LEMMA 82 *Algorithm 9 has time complexity*

$$\begin{aligned}
&(2m+3k+(2k+2n+8)N^m+5)T_{\leftarrow}+(3k+2m+(m+n+4k+3)N^m)T_i\\
&+(2k+3+N^m(2n+1))T_a+N^m(2n-1)T_m+(k+N^m(k+n+2))T_R+N^mT_o\\
&\quad+2T_{p,N,m}+\frac{N(N^m-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})\\
&\quad+\sum_{i=1}^{k}\left(\left(\prod_{j=i}^{k}N^{c_j}\right)(4T_i+2T_a+T_{\leftarrow}+T_R+T_{\mathsf{and}}+T_{a,N^{c_i}})\right)\\
&\quad+\sum_{j=1}^{k}\Bigg(2T_{p,N,c_i}+N^{c_j}((2c_j+1)T_i+T_o\\
&\quad+(c_j+1)T_R+(2c_j+3)T_{\leftarrow}+(2c_j-1)T_a+2(c_j-1)T_m\\
&\quad+\frac{N(N^{c_j}-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})\Bigg),
\end{aligned}$$

which is $\mathcal{O}((m+n+k)N^m)$.

PROOF: Part of the initialization consumes

$$2T_{\leftarrow} + T_{p,N,m} + T_a + 2kT_i + 2kT_{\leftarrow} + kT_a + kT_R + \sum_{i=1}^{k} T_{p,N,c_i} + T_{\leftarrow} + k(T_i + T_{\leftarrow})$$

time units. The conversion of f to F consumes the number of time units given in lemma 76. The conversions of the mappings $h_1, \ldots, h_k$ consume

$$\sum_{j=1}^{k} \Bigg(c_j(T_i + T_{\leftarrow}) + 2T_{\leftarrow} + T_{p,N,c_j} + T_a + N^{c_j}((2c_j+1)T_i + T_o + (c_j+1)T_R + (2c_j+3)T_{\leftarrow} + (2c_j-1)T_a + 2(c_j-1)T_m) + \frac{N(N^{c_j}-1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}}) \Bigg)$$

time units. The main (outer) loop is run N^m times, the inner loop k times. Since the main loop uses all possible executions of algorithm 5, it consumes

$$N^m(2T_{\leftarrow} + k(2T_{\leftarrow} + 4T_i + T_m + 2T_a + T_R) + 2T_{\leftarrow} + 2T_i + T_a + T_R + T_{\leftarrow}) + \sum_{i=1}^{k} \left(\left(\prod_{j=i}^{k} N^{c_j} \right) (4T_i + 2T_a + T_{\leftarrow} + T_R + T_{\text{and}} + T_{a,N^{c_i}}) \right)$$

time units. □.

Algorithm 10

In: The mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1, \ldots, f_n)$ indexed by tuples $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$. The mappings $h_1, \ldots, h_k : \mathbb{Z}_N^{c_i} \longrightarrow \mathbb{Z}_N^{c_i}$, represented as tables containing all N^{c_i} mapping values respectively. The integers $N, k, m, c_1, \ldots, c_k$, such that $\sum_{i=1}^{k} c_i = n$.

Out: The mapping

$$L(x_1,\ldots,x_m) = (h_1(f_1(x_1,\ldots,x_m),\ldots,f_{c_1}(x_1,\ldots,x_m)),\ldots,\\ h_k(f_{n-c_k+1}(x_1,\ldots,x_m),\ldots,f_n(x_1,\ldots,x_m))),$$

represented as a function $L : \mathbb{Z}_{N^m} \longrightarrow \mathbb{Z}_{N^n}$.

$i \leftarrow 1, K \leftarrow N^m - 1$	$2T_{\leftarrow} + T_{p,N,m} + T_a$
do:	
Convert h_i to $H_i : \mathbb{Z}_{N^{c_i}} \longrightarrow \mathbb{Z}_{N^{c_i}}$ using algorithm 6	
$y_i \leftarrow N^{c_i}$	$T_{\leftarrow} + 2T_i + T_{p,N,c_i}$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while($i \leq k$)	T_R
$(x_1,\ldots,x_m) \leftarrow (0,\ldots,0)$	$m(T_i + T_{\leftarrow})$
$i \leftarrow 0$	$T_{\leftarrow}$
do:	
$u \leftarrow 0, j \leftarrow k, a \leftarrow n$	$3T_{\leftarrow}$
do:	
$l \leftarrow c_j, v \leftarrow 0$	$2T_{\leftarrow} + T_i$
do:	
$v \leftarrow Nv + f_a(x_1,\ldots,x_m)$	$(m+1)T_i + T_o + T_m + T_a + T_{\leftarrow}$
$l \leftarrow l - 1$	$T_{\leftarrow} + T_a$
$a \leftarrow a - 1$	$T_{\leftarrow} + T_a$
while($l \geq 1$)	$T_i + T_R$
$u \leftarrow y_j u + H_j(v)$	$T_{\leftarrow} + 3T_i + T_m + T_a$
$j \leftarrow j - 1$	$T_{\leftarrow} + T_a$
while($j \geq 1$)	T_R
$L(i) \leftarrow u$	$T_i + T_{\leftarrow}$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
Increment $\vec{x}$ using algorithm 4	
while($i \leq K$)	T_R

LEMMA 83 *Algorithm 10 has approximate space complexity*

$$\left((2n+4m)N^m + 2\sum_{j=1}^{k}(c_j+1)N^{c_j} + 1 + \max\{c_j\}\right)\lceil\log_2 N\rceil$$
$$\lceil\log_2 k\rceil + \lceil\log_2 m\rceil + 2\lceil\log_2 n\rceil + \sum_{j=1}^{k}\lceil\log_2 c_j\rceil + \lceil\log_2(\max\{k,m\})\rceil$$
$$+ \lceil\log_2(\max\{c_i\})\rceil$$

which is $\mathcal{O}((n+m)N^m)$.

PROOF: The tables for the mappings $f, h_1, \ldots, h_k$ occupy

$$nN^m\lceil\log_2 N\rceil, c_1N^{c_1}\lceil\log_2 N\rceil, \ldots, c_kN^{c_k}\lceil\log_2 N\rceil$$

storage units, respectively. The tables for $H_1, \ldots, H_k, L$ occupy

$$N^{c_1}\lceil\log_2(N^{c_1})\rceil, \ldots, N^{c_k}\lceil\log_2(N^{c_k})\rceil, N^m\lceil\log_2(N^n)\rceil$$

storage units, respectively. The integers $N, k, m, n, c_1, \ldots, c_k, i, j, K, u, v, a, l$ occupy

$$\lceil\log_2(N+1)\rceil, \lceil\log_2(k+1)\rceil, \lceil\log_2(m+1)\rceil, \lceil\log_2(n+1)\rceil, \lceil\log_2(c_1+1)\rceil, \ldots,$$
$$\lceil\log_2(c_k+1)\rceil, \lceil\log_2(N^m)\rceil, \lceil\log_2(\max\{k,m\}+1)\rceil, \lceil\log_2(N^m)\rceil, \lceil\log_2(N^m)\rceil$$
$$\lceil\log_2(N^{\max\{c_i\}})\rceil, \lceil\log_2(n+1)\rceil, \lceil\log_2(\max\{c_i\}+1)\rceil$$

storage units, respectively. The integers $x_1, \ldots, x_m, y_1, \ldots, y_k$ occupy

$$m\lceil\log_2 N\rceil, \lceil\log_2(N^{c_1}+1)\rceil, \ldots, \lceil\log_2(N^{c_k}+1)\rceil$$

storage units, respectively. Algorithm 6 does not need any additional variables. Algorithm 4 doesn't require any additional variables. □.

LEMMA 84 *Algorithm 10 has time complexity*

$$(m+n+4k+3+N^m(3n+4k+5))T_{\leftarrow}+(m+n+2k+N^m(mn+n+4k+1))T_i$$
$$(2k+1+N^m(3n+2k+1))T_a+(k+N^m(n+k+1))T_R+nN^mT_o+(n+k)N^mT_m$$
$$+2\sum_{j=1}^{k}T_{p,N,c_j}+T_{p,N,m}$$
$$+\frac{N(N^m-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})$$
$$\sum_{j=1}^{k}\Bigg(N^{c_j}((2c_j+1)T_i+T_o+(c_j+1)T_R+(2c_j+3)T_{\leftarrow}+(2c_j-1)T_a$$
$$+2(c_j-1)T_m)+\frac{N(N^{c_j}-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})\Bigg),$$

which is $\mathcal{O}(mnN^m)$.

PROOF: Initialization consumes $2T_{\leftarrow}+T_{p,N,m}+T_a+2kT_i+\sum_{j=1}^{k}T_{p,N,c_i}+kT_R+2kT_{\leftarrow}+kT_a+m(T_i+T_{\leftarrow})+T_{\leftarrow}$ time units. The conversions of the mappings $h_1,\ldots,h_k$ consume

$$\sum_{j=1}^{k}\Bigg(c_j(T_i+T_{\leftarrow})+2T_{\leftarrow}+T_{p,N,c_j}+T_a$$
$$+N^{c_j}((2c_j+1)T_i+T_o+(c_j+1)T_R+(2c_j+3)T_{\leftarrow}+(2c_j-1)T_a+2(c_j-1)T_m)$$
$$+\frac{N(N^{c_j}-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})\Bigg)$$

time units. The main (outer) loop is run N^m times, the next nested loop k times, and the innermost loop c_j times. Thus, for each iteration of the outer loop, the innermost loop is run $n$times. Also, each possible run of algorithm 4 occurs once and only once during the completion of the outer

loop. The main loop thus consumes

$$N^m(5T_{\leftarrow} + T_i + T_a + T_R + n(3T_{\leftarrow} + (m+1)T_i + T_o + T_m + 3T_a + T_R) + k(4T_{\leftarrow} + 4T_i + 2T_a + T_m + T_R)) + \frac{N(N^m-1)}{(N-1)}(3T_i + T_{a,N} + 2T_a + 2T_{\leftarrow} + 2T_R + T_{\text{and}})$$

time units. □.

Algorithm 11

In: The mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1, \ldots, f_n)$ indexed by tuples $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$. The mappings $h_1, \ldots, h_k : \mathbb{Z}_N^{d_i} \longrightarrow \mathbb{Z}_N^{c_i}$, represented as tables containing all N^{d_i} mapping values respectively. The integers $m, k, n, c_1, \ldots, c_k$, such that $\sum_{i=1}^k c_i = m$. The functions $e(i, \cdot) : \mathbb{Z}_{d_i} \longrightarrow \{1, \ldots, m\}$ for $i \in \{1, \ldots, k\}$.

Out: The mapping $F : \mathbb{Z}_N^{|\cup_{(i,j)} e(i,j)|} \longrightarrow \mathbb{Z}_N^n$ defined by the composition

$$f(h_1(x_{e(1,1)}, \ldots, x_{e(1,d_1)}), \ldots, h_k(x_{e(k,1)}, \ldots, x_{e(k,d_k)}))$$

represented as a function table.

$i \leftarrow 1$	$T_{\leftarrow}$
do:	
$u(i) \leftarrow 0$	$T_{\leftarrow} + T_i$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while($i \leq m$)	T_R
$i \leftarrow 1, U \leftarrow 0$	$2T_{\leftarrow}$
do:	
$j \leftarrow 1$	$T_{\leftarrow}$
do:	
if $u(e(i,j)) = 0$ then	$T_R + T_o + 3T_i$
$U \leftarrow U + 1$	$T_{\leftarrow} + T_a$

$u(e(i,j)) \leftarrow U$	$T_{\leftarrow} + 3T_i + T_o$
$j \leftarrow j + 1$	$T_{\leftarrow} + T_a$
while($j \leq d_j$)	$T_R + T_i$
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while($i \leq k$)	T_R
$i \leftarrow 0, K \leftarrow N^U - 1$	$2T_{\leftarrow} + T_{p,N,U} + T_a$
$(a_1, \ldots, a_U) \leftarrow (0, \ldots, 0), a_0 \leftarrow 0$	$(U+1)(T_i + T_{\leftarrow})$
do:	
$j \leftarrow 1$	$T_{\leftarrow}$
do:	
$b_j \leftarrow a(u(j))$	$T_{\leftarrow} + 3T_i$
$j \leftarrow j + 1$	$T_{\leftarrow} + T_a$
while($j \leq m$)	T_R
$j \leftarrow 1, b \leftarrow 1$	$2T_{\leftarrow}$
do: $l \leftarrow 1$	$T_{\leftarrow}$
do:	
$x_b \leftarrow h_{j,l}(b_{e(j,1)}, \ldots, b_{e(j,d_j)})$	$d_j(T_o + 3T_i) + T_o + 4T_i + T_{\leftarrow}$
$b \leftarrow b + 1$	$T_{\leftarrow} + T_a$
$l \leftarrow l + 1$	$T_{\leftarrow} + T_a$
while($l \leq c_j$)	$T_R + T_i$
while($j \leq k$)	T_R
$F(a_1, \ldots, a_U) \leftarrow f(x_1, \ldots, x_m)$	$(U+m)T_i + 2T_o + nT_{\leftarrow}$
Increment $(a_1, \ldots, a_U)$ using algorithm 4	
$i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while($i \leq K$)	T_R

LEMMA 85 *Algorithm 11 has approximate space complexity*

$$\left(nN^m + 3U + 2m + \sum_{j=1}^{k} c_j N^{d_j} + nN^U\right) \lceil \log_2 N \rceil + \lceil \log_2 U \rceil + \lceil \log_2(\max\{c_j\}) \rceil$$
$$+ \left(m + 2 + \sum_{j=1}^{k} d_j\right) \lceil \log_2 m \rceil + 2\lceil \log_2 k \rceil + \lceil \log_2 n \rceil + \sum_{j=1}^{k} (\lceil \log_2 c_j \rceil + \lceil \log_2 d_j \rceil)$$

which is $\mathcal{O}\left(\left(nN^m + \sum_{j=1}^{k} c_j N^{d_j}\right) \lceil \log_2 N \rceil\right)$.

PROOF: The tables for the mappings $f, h_1, \ldots, h_k, e$ occupy

$$nN^m \lceil \log_2 N \rceil, c_1 N^{d_1} \lceil \log_2 N \rceil, \ldots, c_k N^{d_k} \lceil \log_2 N \rceil, \sum_{j=1}^{k} d_j \lceil \log_2 m + 1 \rceil$$

storage units, respectively. The mapping F occupies $nN^U \lceil \log_2 N \rceil$ storage units. The integers $m, k, n, c_1, \ldots, c_k, d_1, \ldots, d_k, i, j, l, U, K, b$ occupy

$$\lceil \log_2(m+1) \rceil, \lceil \log_2(k+1) \rceil, \lceil \log_2(n+1) \rceil, \lceil \log_2(c_1+1) \rceil, \ldots, \lceil \log_2(c_k+1) \rceil,$$
$$\lceil \log_2(d_1+1) \rceil, \ldots, \lceil \log_2(d_k+1) \rceil, \lceil \log_2(N^U) \rceil, \lceil \log_2(k+1) \rceil,$$
$$\lceil \log_2(\max\{c_j\}+1) \rceil, \lceil \log_2(U+1) \rceil, \lceil \log_2(N^U) \rceil, \lceil \log_2(m+1) \rceil$$

storage units, respectively. The vectors $(x_1, \ldots, x_m)$, $(b_1, \ldots, b_m)$ $(a_1, \ldots, a_U)$, and the array $u(1), \ldots, u(m)$ occupy

$$m\lceil \log_2 N \rceil, m\lceil \log_2 N \rceil, U\lceil \log_2 N \rceil, m\lceil \log_2(m+1) \rceil$$

storage units, respectively. Algorithm 4 has no need of any additional local variables $\square$.

LEMMA 86 *Algorithm 11 has time complexity*

$$(2m+2k+U+6+N^U(5m+n+k+5))T_{\leftarrow}+(m+U+1+N^U(9m+U))T_i$$
$$(m+k+N^U(2m+k+1))T_R+(m+k+1+N^U(3m+1))T_a+N^U(m+2)T_o$$
$$+\sum_{j=1}^{k} d_j(3T_{\leftarrow}+2T_R+(2+N^U)T_o+2T_a+(7+3N^U)T_i)+T_{p,N,U}$$
$$+\frac{N(N^U-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})$$

which is $\mathcal{O}((m+k+U)N^U+N^U\sum_{j=1}^{k} d_j)$.

PROOF: Initialization consumes

$$(2m+2k+U+6)T_{\leftarrow}+(m+U+1)T_i+(m+k)T_R+(m+k+1)T_a$$
$$\sum_{j=1}^{k} d_j(3T_{\leftarrow}+2T_R+2T_o+2T_a+7T_i)+T_{p,N,U}$$

time units. Every possible run of algorithm 4 has been executed by the time the main loop finishes. Assuming that the if always evaluates to true (the closest one can get to a conservative assumption), it follows that the main loop consumes:

$$N^U((5m+n+k+5)T_{\leftarrow}+(9m+U)T_i+(2m+k+1)T_R+(m+2)T_o$$
$$(3m+1)T_a+\sum_{j=1}^{k} d_j(T_o+3T_i))$$
$$+\frac{N(N^U-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})$$

□.

Algorithm 12

In: The mapping $f : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^n$ represented as a table containing all N^m mapping values $(f_1, \ldots, f_n)$ indexed by tuples $(x_1, \ldots, x_m) \in \mathbb{Z}_N^m$. The mappings $h_1, \ldots, h_k : \mathbb{Z}_N^{c_i+d_i} \longrightarrow \mathbb{Z}_N^{b_i}$, represented as tables containing all N^{d_i} mapping values respectively. The integers $m, n, o, k, c_1, \ldots, c_k$, such that $\sum_{i=1}^{k} c_i = n$ and $\sum_{i=1}^{k} b_i = o$. The functions $e(i, \cdot) : \mathbb{Z}_{d_i} \longrightarrow \{1, \ldots, m\}$ for $i \in \{1, \ldots, k\}$. The functions $e'(i, \cdot) : \mathbb{Z}_{c_i} \longrightarrow \{1, \ldots, n\}$ for $i \in \{1, \ldots, k\}$.

Out: The mapping $F : \mathbb{Z}_N^m \longrightarrow \mathbb{Z}_N^o$ defined by the composition

$$(h_1(f_{e'(1,1)}(x_1, \ldots, x_m), \ldots, f_{e'(1,c_1)}(x_1, \ldots, x_m), x_{e(1,1)}, \ldots, x_{e(1,d_1)}), \ldots,$$
$$h_k(f_{e'(k,1)}(x_1, \ldots, x_m), \ldots, f_{e'(k,c_k)}(x_1, \ldots, x_m), x_{e(k,1)}, \ldots, x_{e(k,d_k)})).$$

$i \leftarrow 0, K \leftarrow N^m$	$2T_{\leftarrow} + T_{p,N,m}$
$(a_1, \ldots, a_m) \leftarrow (0, \ldots, 0)$	$m(T_i + T_{\leftarrow})$
do:	
$\quad j \leftarrow 1, a \leftarrow 1$	$2T_{\leftarrow}$
$\quad$ do:	
$\quad\quad l \leftarrow 1$	$T_{\leftarrow}$
$\quad\quad$ do:	
$\quad\quad\quad x_a \leftarrow h_{j,l}(f_{e'(j,1)}(\vec{a}), \ldots, f_{e'(j,c_j)}(\vec{a}), a_{e(j,1)}, \ldots, a_{e(j,d_j)})$	$(m + 3c_j + 3d_j)T_i + T_o + 4T_i + T_{\leftarrow}$
$\quad\quad\quad a \leftarrow a + 1$	$T_{\leftarrow} + T_a$
$\quad\quad\quad l \leftarrow l + 1$	$T_{\leftarrow} + T_a$
$\quad\quad$ while($l \leq b_j$)	$T_R + T_i$
$\quad\quad j \leftarrow j + 1$	$T_{\leftarrow} + T_a$
$\quad$ while($j \leq k$)	T_R
$\quad F(a_1, \ldots, a_m) \leftarrow (x_1, \ldots, x_o)$	$(m + o)T_i + 2T_o + T_{\leftarrow}$
$\quad$ increment $(a_1, \ldots, a_m)$ using algorithm 4	
$\quad i \leftarrow i + 1$	$T_{\leftarrow} + T_a$
while($i < K$)	T_R

LEMMA 87 *Algorithm 12 has approximate space complexity*

$$\left(2nN^m + 3m + o + \sum_{j=1}^{k}(b_j N^{c_j+d_j})\right)\lceil\log_2 N\rceil + \lceil\log_2 o\rceil + \lceil\log_2 k\rceil$$
$$+ \left(1 + \sum_{j=1}^{k} d_j\right)\lceil\log_2 m\rceil + \left(1 + \sum_{j=1}^{k} c_j\right)\lceil\log_2 n\rceil$$
$$+ \sum_{j=1}^{k}(\lceil\log_2 b_j\rceil + \lceil\log_2 c_j\rceil + \lceil\log_2 d_j\rceil) + \lceil\log_2(\max\{b_j\})\rceil$$

which is $\mathcal{O}(nN^m + \sum_{j=1}^{k}(b_j N^{c_j+d_j}))$.

PROOF: The tables for the mappings $f, h_1, \ldots, h_k, e, e', F$ occupy

$$nN^m\lceil\log_2 N\rceil, b_1 N^{c_1+d_1}\lceil\log_2 N\rceil, \ldots, b_k N^{c_k+d_k}\lceil\log_2 N\rceil,$$
$$\sum_{j=1}^{k} d_j\lceil\log_2 m+1\rceil, \sum_{j=1}^{k} c_j\lceil\log_2 n+1\rceil, nN^m\lceil\log_2 N\rceil$$

storage units, respectively. The integers $m, k, n, c_1, \ldots, c_k, d_1, \ldots, d_k$ occupy

$$\lceil\log_2(m+1)\rceil, \lceil\log_2(k+1)\rceil, \lceil\log_2(n+1)\rceil, \lceil\log_2(c_1+1)\rceil, \ldots, \lceil\log_2(c_k+1)\rceil,$$
$$\lceil\log_2(d_1+1)\rceil, \ldots, \lceil\log_2(d_k+1)\rceil$$

storage units, respectively. The integers $b_1, \ldots, b_k, i, j, l, K, a$ occupy

$$\lceil\log_2(b_1+1)\rceil, \ldots, \lceil\log_2(b_k+1)\rceil, \lceil\log_2(N^m)\rceil, \lceil\log_2(k+1)\rceil,$$
$$\lceil\log_2(\max\{b_j\}+1)\rceil, \lceil\log_2(N^m)\rceil, \lceil\log_2(o+1)\rceil$$

storage units, respectively. The vectors $(x_1, \ldots, x_o)$ and $(a_1, \ldots, a_m)$ occupy $o\lceil\log_2 N\rceil$ and $m\lceil\log_2 N\rceil$ storage units, respectively. Algorithm 4 does not require additional storage $\square$.

LEMMA 88 *Algorithm 12 has time complexity*

$$(2+m+N^m(3o+2k+5))T_{\leftarrow}+(m+N^m(m+6o+mo))T_i+N^m(k+o+1)T_R$$
$$N^m(k+2o+1)T_a+N^m(o+2)T_o+3N^m\sum_{j=1}^{k}b_j(c_j+d_j)T_i$$
$$+\frac{N(N^m-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}}),$$

which is $\mathcal{O}(moN^m)$.

PROOF: Initialization consumes $2T_{\leftarrow}+T_{p,N,m}+m(T_i+T_{\leftarrow})$ time units. Every possible run of algorithm 4 is executed during the main loop. Thus the main loop consumes

$$N^m((3o+2k+5)T_{\leftarrow}+(m+6o+mo)T_i+(k+o+1)T_R+(k+2o+1)T_a$$
$$+(o+2)T_o+3\sum_{j=1}^{k}b_j(c_j+d_j)T_i)$$
$$+\frac{N(N^m-1)}{(N-1)}(3T_i+T_{a,N}+2T_a+2T_{\leftarrow}+2T_R+T_{\mathsf{and}})$$

□.

Bibliography

[1] Martín Abadi and Joan Feigenbaum. Secure circuit evaluation: A protocol based on hiding information from an oracle. *Journal of Cryptology*, (2):1–12, 1990. 1.1.6

[2] Martín Abadi, Joan Feigenbaum, and Joe Kilian. On hiding information from an oracle (full version). In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, 1987. Proceedings contains only extended abstract—the complete paper was utilized for this dissertation. 1.1.6

[3] David Aucsmith and Gary Graunke. Tamper resistant methods and apparatus. US Patent # 5,892,899, Filed June 13, 1996. 1.1.13

[4] Donald Beaver and Joan Feigenbaum. Hiding instances in multioracle queries. In C. Choffrut and T. Lengauer, editors, *Proceedings of the 7th Annual Symposium on Theoretical Aspects of Computer Science*, volume 415 of *Lecture Notes in Computer Science*. Springer-Verlag, February 1990. 1.1.6

[5] Robert M. Best. Crypto microprocessor using block cipher. US Patent # 4,319,079, Filed January 17, 1980. 1.1.2

[6] Robert M. Best. Crypto microprocessor that executes enciphered programs. US Patent # 4,465,901, Filed July 2, 1981.

[7] Robert M. Best. Crypto microprocessor for executing enciphered programs. US Patent # 4,278,837, Filed June 4, 1979.

[8] Robert M. Best. Microprocessor for executing enciphered programs. US Patent # 4,168,396, Filed October 31, 1977.

[9] Robert M. Best. Cryptographic decoder for computer programs. US Patent # 4,433,207, Filed September 10, 1981. 1.1.2

[10] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: *NP*-completeness, recursive functions, and universal machines. *Bulletin of the American Mathematical Society*, 21(1):1–46, July 1989. 2.2, 2.2.1, 2.2.1, 2.2.3

[11] Dan Boneh and Richard Lipton. Algorithms for black-box fields and their application to cryptography. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO'96*, Lecture Notes in Computer Science, page 283 ff. Springer-Verlag, 1996. 1.1.1, 1.1.10, 6

[12] Gilles Brassard and Claude Crepeau. Zero-knowledge simulation of boolean circuits. In A. M. Odlyzko, editor, *Advances in Cryptology—CRYPTO'86: Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 223–233. Springer-Verlag, 1986. 1.1.4

[13] Ernest F. Brickell and Yacov Yacobi. On privacy homomorphisms (extended abstract). In D. Chaum and W.L. Price, editors, *Advances in Cryptology—Eurocrypt '87*, Lecture Notes in Computer Science, pages 117–125. Springer-Verlag, 1987. 1.1.1

[14] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous agents. In U. Montanari et.al., editor, *ICALP 2000*, Lecture Notes in Computer Science, pages 512–523. Springer-Verlag, 2000. Extended Abstract. 1.1.9

[15] John F. Canny, Erich Kaltofen, and Lakshman Yagati. Solving systems of non-linear polynomial equations faster. In *Proceedings of the*

International Symposium on Symbolic Algebraic Computation (ISSAC'92), pages 121–128. ACM Press, 1992. C

[16] David Chaum, Ivan B. Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO'87: Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 87–119. Springer-Verlag, 1987. 1.1.5

[17] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1983 reprint edition, 1983. 4.1, 4.1

[18] Matthew Dickerson. *The Functional Decomposition of Polynomials*. PhD thesis, Cornell University, 1989. 4.4.7, 4.4.7, 4.4.7, 4.7.7

[19] John J. Glover. Computer system and process for accessing an encrypted and self-decrypting digital information product while restricting access to decrypted digital information. US Patent # 6,052,870, Filed July 3, 1997. 1.1.13

[20] Stuart Alan Haber. *Multi-party Cryptographic Computation: Techniques and Applications*. PhD thesis, Columbia University, 1988. 1.1.3

[21] Bradford E. Hampson. Digital computer system for executing encrypted programs. US Patent # 4,847,902, Filed February 10, 1984. 1.1.2

[22] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1979. 2.1

[23] Dexter Kozen and Susan Landau. Polynomial decomposition algorithms. *Journal of Symbolic Computation*, pages 445–456, 1989. 4.4.7

[24] Serge Lang. *Algebra*. Addison-Wesley Publishing Company, Inc., third edition, 1993. B.2

[25] Sergio Loureiro. *Mobile Code Protection*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2001. 1.1.12

[26] Robert M. Lumley. Digital computer having code conversion apparatus for an encrypted program. US Patent # 4,306,289, Filed February 4, 1980. 1.1.2

[27] Richard L. Maliszewski. Cell array providing non-persistent secret storage through a mutation cycle. US Patent # 6,049,609, Filed August 6, 1997. 1.1.13

[28] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. 1.1.4

[29] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In Richard A. DeMillo, editor, *Foundations of Secure Computation*. Academic Press, 1978. 1.1.1, 1.1.1, 4.1

[30] Yurii Rogozhin. Small universal turing machines. *Theoretical Computer Science*, (168):215–240, 1996. 1, 3.4, 3.5, 3.6

[31] Tomas Sander and Christian F. Tschudin. Towards mobile cryptography. Technical report, International Computer Science Institute, November 1997. 1.1.7

[32] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In Giovanni Vigna, editor, *Mobile Agents and Security*, LNCS State-of-the-Art Survey. Springer-Verlag, 1998. 1.1.7, 1.1.12, 4.1

[33] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for NC^1. In 40^{th} *Annual Symposium on Foundations of Computer Science*, pages 554–566. IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1999. 1.1.8

[34] Claude Elwood Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, pages 656–715, 1948. 4.3, 4.4.3, 4.6, 4.7.3, 4.9.3

[35] Andrew C. Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, volume 23. IEEE Computer Society's Technical Committee on Mathematical Foundations of Computing, November 1982. 1.1.3, 1.1.3