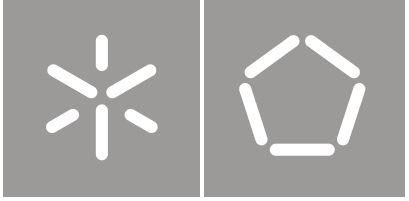
**Universidade do Minho**
Escola de Engenharia

Nuno Tiago Ferreira de Carvalho

# A practical validation of Homomorphic Message Authentication schemes

Outubro de 2014



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Nuno Tiago Ferreira de Carvalho

# A practical validation of Homomorphic Message Authentication schemes

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

**Professor Manuel Bernardo Barbosa**

Outubro de 2014

## Abstract

Currently, cloud computing is very appealing because it allows the user to outsource his data so it can later be accessed from multiple devices. The user can also delegate to the cloud computing service provider some, possibly complex, operations on the outsourced data. Since this service provider may not always be trusted, it is necessary to not only preserve the privacy but also to enforce the authenticity of the outsourced data. Lately, a lot of work was put on solving the first problem, specially after the introduction of the first Fully Homomorphic Encryption scheme. In this work we will focus on the latter, namely on the use of Homomorphic Message Authentication primitives. We will evaluate the current available solutions, their functionality and their security. Finally, we will provide an implementation of one of these schemes in order to verify if they are indeed practical.

# Acknowledgements

To my supervisor, Professor Manuel Bernardo Barbosa, for the great opportunity, all the guidance and all the suggestions that lead me here, many thanks.

To Professor Dario Catalano, for receiving me in Catania and for all the precious help and advises given during my stay. A special thanks to Professor Mario di Raimondo for the practical suggestions.

To all my friends, specially the ones who kept up with me during these last few stressing weeks. A special mention to the ones who somehow were with me during these last 6 years of studying.

To my friends in Catania, specially the people from the residenza, for understanding the reason of my absence so many times.

Lastly, but not least, to my parents, Domingos and Maria, and to my sister, Joana, without whom I wouldn't be here, a profound and sincere thank you for providing me the with the necessary conditions and support to be here now.

Este trabalho foi apoiado pelo Projeto Smartgrids - NORTE-07-0124-FEDER-000056, cofinanciado pelo Programa Operacional Regional do Norte (ON.2 – O Novo Norte), ao abrigo do Quadro de Referência Estratégico Nacional (QREN), através do Fundo Europeu de Desenvolvimento Regional (FEDER).

# Contents

# Abbreviations

**CRT** Chinese Remainder Theorem

**DAG** Directed Acyclic Graph

**DFS** Depth-First Search

**DFT** Discrete Fourier Transform

**FFT** Fast Fourier Transform

**FHE** Fully Homomorphic Encryption

**MAC** Message Authentication Code

**OWF** One-Way Function

**PPT** Probabilistic-Polynomial Time

**PRF** Pseudo-Random Function

**SIS** Small Integer Solution

**SIVP** Shortest Independent Vectors Problem

**VC** Verifiable Computation

# Chapter 1

# Introduction

A cloud computing provider allows a user to lease computing and storage resources from remote computers that are far more powerful than the user's. This enables her to perform operations over (some portion of) outsourced data that is remotely stored and then only the results of these operations are returned to the user. However, the current infrastructures of cloud computing services do not provide any security against untrusted cloud operators, making them unsuitable for storing sensitive information such as medical, financial or any personal records.

A possible solution to this problem is the use of homomorphic cryptography. This concept was first presented by Rivest, Adleman, and Dertouzos [RAD78] about 30 years ago. However, it remained an open problem until recently, when the first truly homomorphic schemes were proposed. With these schemes it became possible to perform computations on encrypted data such that the cloud computing provider does not have access to the decrypted information, therefore preserving *privacy* of the outsourced data. The first FHE scheme was proposed in a ground-breaking work by Gentry [Gen09]. His scheme allows the computation of arbitrary computations over encrypted data without being able to decrypt. Unfortunately it is not yet practical, but several small improvements have been proposed which could lead it to be practical in the near future.

But, and not only in the context of cloud computing, it is also important to guarantee the *authenticity* of the computations performed on the outsourced data. FHE schemes by itself only guarantee privacy, and not authenticity. To do so, two main goals need to be fulfilled: *security* and *efficiency*. Security guarantees that the server is able to "prove" the correctness of the delegated computation. Efficiency means that the user is able to efficiently check the proof by requiring far less resources than those needed to execute the delegated computation (including both computation and communication).

## 1.1 Motivation

To guarantee authenticity for computations on outsourced data a new set of homomorphic primitives were defined: homomorphic signatures (asymmetric setting) and homomorphic message authenticators (symmetric setting).

The first notions of homomorphic signatures were introduced around 15 years ago by Rivest et al. [RMCR01], and then Johnson et al. [JMSW02] presented a more complete definition. An homomorphic signature scheme allows a user to generate signatures $\sigma_1, \ldots, \sigma_k$ on messages $m_1, \ldots, m_k$ so that later anyone (without the private signing key) can compute a signature $\sigma'$ that is valid for the value $m' = f(m_1, \ldots, m_k)$. Also, it allows anyone that holds the public verification key to verify the results. Boneh and Freeman [BF11] constructed a scheme that evaluates multivariate polynomials on previously signed data, but they also stated that the construction of a fully homomorphic signatures scheme remains an open problem.

The secret key analogue of homomorphic signatures are homomorphic message authenticators (homomorphic MACs for short) that were introduced by Gennaro and Wichs [GW12]. In this setting, there is a public evaluation key and a secret key shared by Alice and Bob, which is used to authenticate some messages $m_1, \ldots, m_k$. Then, anybody who has the public evaluation key can compute a short tag $\sigma$ that certifies the value $y = f(m_1, \ldots, m_k)$ as the output of $f$. Notice that $\sigma$ does not simply authenticate $y$ out of context, it only certifies it as the output of a specific $f$. Later, Bob can verify that $y$ is indeed the output of $f(m_1, \ldots, m_k)$ without even knowing the value of the messages. Gennaro and Wichs [GW12] also presented a construction with support for arbitrary computations, a fully homomorphic MAC, but its security is based on a weaker model (the adversary cannot ask for verification queries, only for authentication queries).

To overcome this problem, and relying on much of the work from Gennaro and Wichs, Catalano and Fiore [CF13] constructed a very simple and efficient scheme with support for a more restricted class of computations. Furthermore, its security is based on a stronger model with support for multiple verification queries. But both of these constructions suffered from a common problem: the verification algorithm runs in time proportional to the size of the function. The scheme from Backes, Fiore, and Reischuk [BFR13] tries to solve this problem, and it is the first homomorphic MAC scheme with efficient (amortized) verification.

## 1.2 Objectives

We intend to study the current homomorphic MAC constructions in order to better understand their strengths and pitfalls. Having realized what those are, we want to implement one of these constructions, Catalano and Fiore [CF13] to be more precise. The reason we ignore the implementation of Gennaro and Wichs [GW12] is because of its reduced security and because it relies heavily on lattice based techniques, which are not yet efficient.

Our main goal is then to measure the overhead introduced by the homomorphic evaluation of [CF13]. In practical terms, what we intend to measure is the amount of extra work that the external server must do to homomorphically compute the function in comparison to a normal computation.

We will also measure the complexity of all the other operations of Catalano and Fiore [CF13] in order to check if it is already practical.

## 1.3 Structure of the document

This document is split in seven chapters. Chapter 1 is an introduction to the problem of homomorphic authentication and describes what is the purpose of this dissertation.

**Chapter 2** presents a brief review of the current main techniques used for delegation of computations.

**Chapter 3** presents an alternative model of computation based on circuits. We start by defining what are circuits, and we explain how they can be represented and how can we evaluate them. We also present two conversion tools that convert `C` into arithmetic circuits.

**Chapter 4** describes the analysis we made to the homomorphic authentication primitives. We start by introducing the necessary preliminaries and definitions of homomorphic signatures and MACs, and then by presenting the actual constructions for both settings. In the end we state what the current limitations and open problems are in both settings.

**Chapter 5** introduces the main algebraic operations necessary for our work. The main point of this chapter is the study of an efficient polynomial multiplication algorithm based on FFT.

**Chapter 6** presents our experimental results. We show here how our implementation performs.

Finally, in Chapter 7 we present the final conclusions of all the work we developed, as well as what can be done in the future.

# Chapter 2

# Delegation of computations

Currently, if Alice wants do delegate a program $f$ to an external server (as in "cloud computing") she has to store all the data unencrypted on the external server, and then the server can perform the computation and send the result back to Alice. If the server is not trusted there is no way that Alice can verify that the output she received is indeed correct, or as importantly than this, Alice's privacy is not enforced since the external server has full access to her data. A very recent scandal showed that governmental agencies may have included backdoors in many famous cloud services that would allow them to easily have access to the users outsourced data[1].

In a perfect world where it is possible for Alice to delegate any computation $f$ over a data set of arbitrary size, she just sends the encrypted data to an external server, along with a description of the computation $f$, and then the server executes $f$ over the encrypted data and returns to Alice the encrypted result of $f$. Afterwards, she can obtain the unencrypted result using her secret key. This technique would enforce the *privacy* of the outsourced data set, but this would not give Alice the possibility of verifying the correctness of results. To do so, the server would also be able to send to Alice some small proof that she could verify very easily, and therefore there would a guarantee of *authenticity*.

Unfortunately, there are not yet any practical solutions to achieve this "perfect" scenario. And in the current world, there is no privacy nor any way to easily verify that the output is correct. There were however many advances and new techniques introduced in the last few years in order to improve this, and we will briefly introduce some of them.

[1] See `http://www.theguardian.com/world/2013/jun/06/nsa-phone-records-verizon-court-order` or `http://www.theguardian.com/us-news/the-nsa-files`.

## 2.1 Homomorphic encryption

Homomorphic encryption is known to exist for a long time [RAD78]. One of the first realizations was that RSA [RSA78] was partially homomorphic: given two messages $m_1, m_2$ and their corresponding encryptions with RSA, $c_1 = \mathsf{Enc}(m_1), c_2 = \mathsf{Enc}(m_2)$, it holds that $c_1 \cdot c_2 = \mathsf{Enc}(m_1 \cdot m_2)$, i.e., the product of two ciphertexts is the same as the encryption of the product of two messages.

Other popular partially homomorphic schemes include ElGamal [Elg85], Paillier [Pai99] and Boneh-Goh-Nissin [BGN05]. Initially, ElGamal was known to be only partially multiplicatively homomorphic, but later an additively homomorphic variant was proposed [CGS97]. Paillier is both additively and multiplicatively homomorphic. Boneh-Goh-Nissin has the interesting property that allows it to perform multiple additions over ciphertexts, and one final multiplication. A very useful practical use for such partially homomorphic encryption schemes is electronic voting (e-voting) [Adi08] [2].

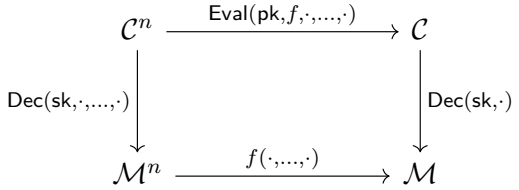
However, these partially homomorphic encryption schemes only support additions and multiplications, so Alice cannot yet delegate the computation of some arbitrary program $f$. This was actually an open problem until very recently when Gentry [Gen09] presented the first realization of a Fully Homomorphic Encryption (FHE) scheme that allows for an unlimited number of computations over encrypted data, just like we have in our perfect world. But as we are not in this perfect world, FHE is still very far from being practical, even tough many improvements to Gentry's original work have been proposed in the last few years.

A FHE scheme is a public-key encryption scheme, and so it is composed of the usual $\mathsf{KeyGen}$, $\mathsf{Enc}$ and $\mathsf{Dec}$ algorithms. The message and ciphertext spaces of the scheme are $\mathcal{M}$ and $\mathcal{C}$ respectively. A ciphertext $c \in \mathcal{C}$ encrypts a message $m \in \mathcal{M}$ under key $(\mathsf{pk}, \mathsf{sk})$, and $\mathsf{Dec}(\mathsf{sk}, c)$ returns $m$.

The extra feature of a FHE scheme is that it comes equipped with an extra *efficient* algorithm *Evaluate*, denoted as $\mathsf{Eval}$. Basically, for every valid key pair $(\mathsf{pk}, \mathsf{sk})$, any $n$ encryptions $c_1, \ldots, c_n \in \mathcal{C}$ of any messages $m_1, \ldots, m_n \in \mathcal{M}$ under $(\mathsf{pk}, \mathsf{sk})$, and for any $n$-ary function $f\colon \mathcal{M}^n \to \mathcal{M}$, $\mathsf{Eval}(\mathsf{pk}, f, c_1, \ldots, c_n)$ outputs a ciphertext $c$ that encrypts $f(m_1, \ldots, m_n)$, i.e., outputs the result of $\mathsf{Enc}(\mathsf{pk}, f(m_1, \ldots, m_n))$ without access to $m_1, \ldots, m_n$. Essentially, $\mathsf{Eval}$ is a public algorithm that anyone can execute without the secret key, and it works like an "impenetrable box" of encryption that executes $f$ inside itself.

The following commutative diagram describes a FHE scheme.

[2] `https://vote.heliosvoting.org/`




As we can see, decryption and application of $f$ is exactly the same as Eval and decryption: either way, the end result is $f(m_1, \ldots, m_n)$.

With such a FHE scheme, Alice can now execute programs on her encrypted data saved on an external server. She simply sends the description of $f$, and the server simply sends her the output of Eval. More over, she can also encrypt the description of the program $f$ so that the external doesn't know which program she is executing nor over which data in particular. The practical applications of such a scheme are enormous: we can send an encrypted query to a search engine, and received the encrypted results; or we can securely store a huge data set, and then compute a complex function over it and obtain an encrypted result. But unfortunately, there isn't yet any practical FHE scheme

Current FHE constructions are all built around the same idea: when encrypting a message, some noise is added to the resulting ciphertext, so that the ciphertexts become "noisy", and computing over them increases the noise, so that eventually it becomes too big that decryption is impossible. To overtake this problem, Gentry introduced the notion of *bootstrapping* so that it is possible to "refresh" the ciphertexts in such a way that they can used for an unbounded number of computations. But this is also the main bottleneck of current constructions of FHE.

In practical terms, a Homomorphic Encryption Library (HELib[3]) was recently released with the purpose of making homomorphic encryption a reality. Unfortunately, and because of the complexity of bootstrapping, only low-level operations are supported.

## 2.2 Homomorphic authentication

In the current (non-homomorphic) cryptographic protocols, in order to enforce the authenticity of a message, i.e., to prove that it wasn't tampered or forged in some way, we need to rely on authentication schemes such as Signatures or Message Authentication Codes (MACs). It would only be natural to have equivalent schemes on a homomorphic scenario, so that when Alice uses a homomorphic encryption scheme, she would receive from the server both the output of the computation as well as some small proof that she can use to verify the authenticity of the obtained output. As in a non-homomorphic scenario, there are two homomorphic authentication primitives – Signatures and MACs.

[3] `https://github.com/shaih/HElib`

These authentication primitives must be both *secure* and *efficient*, meaning that they should be able to prove the correctness of the delegated computation, and the verification should be far less complex than the delegation and computation, i.e., in terms of communication and computation. But what makes these primitives non-trivial is that the client does not keep a local copy of her data, so it rules out trivial solutions such as one where the client performs the computation herself, and then compares the output with the one obtained from the server. Another nice property that homomorphic authenticators should enjoy is *composability*, which means that derived signatures should be usable as inputs for future computations.

**Homomorphic Signatures** The idea of a *homomorphic signature* was first introduced in a series of talks by Rivest et al. [RMCR01], and then formally defined by Johnson et al. [JMSW02]. A homomorphic signature scheme allows us to perform computations over signed data, and, besides the regular signature algorithms KeyGen, Sign and Vrfy, it is augmented with an *evaluation* algorithm Eval that performs the homomorphic computation over the signed data, and then returns a short signature. Basically, if Alice has a data set $m_1, \ldots, m_n$ of size $n$, she must first independently sign each message $m_i$ with her private signing key to obtain $n$ independent signatures $\sigma_1, \ldots, \sigma_n$, and then she stores both the data set and the $n$ signatures on some external server. Later, anyone can ask the server to compute a signature $\sigma'$ that is valid for $m' = f(m_1, \ldots, m_n)$, without ever revealing the original data set, and anyone with access to the public verification key can verify that the server correctly applied $f$ to the data set by verifying the signature $\sigma'$. This derived signature authenticates both the function $f$ and the result of applying $f$ to the data set.

Informally, the security property of a homomorphic signature states that adversaries that can (adaptively) see the signatures corresponding to polynomially many messages of their choice, cannot forge a signature for a new $m^\star \neq f(m_1, \ldots, m_n)$.

The first constructions supported only the computation of linear functions over signed data, and were specially tailored for network coding routing mechanism. The only non-linear construction is the one of Boneh and Freeman [BF11] with support for polynomial functions. The security of BF is based on hard problems on ideal lattices, and it is performed in the random oracle model.

**Homomorphic MACs** In the private key setting we have *homomorphic MACs*, which do not have the property of public verifiability, since only the holders of the private key can perform the verification. The notion was introduced by Gennaro and Wichs [GW12]. With a homomorphic MAC scheme, anyone who holds the public evaluation key and doesn't know the secret key can compute a short MAC that validates some computation over previously authenticated data. The owner of the secret

key can then verify the results of the computation with ever knowing the original authenticated inputs. Just as with homomorphic encryption and signatures, a homomorphic MAC scheme is a regular non-homomorphic MAC scheme augmented with an *evaluation* algorithm Eval to perform the homomorphic computation over the authenticated data set.

Basically, Alice has a secret key that she uses to authenticate a message $m$ to obtain a MAC $\sigma$. Later, given MACs $\sigma_1, \ldots, \sigma_n$ authenticating messages $m_1, \ldots, m_n$, anyone can run a program $f$ over $\sigma_1, \ldots, \sigma_n$ to generate a short MAC that authenticates the output of $f$ over the original messages $m_1, \ldots, m_n$, and only the holders of the secret key can verify the results of such MAC.

Gennaro and Wichs also presented a construction with support for arbitrary computations, a Fully Homomorphic MAC, but its security is based on a weaker model where an adversary cannot ask for verification queries, but only for authentication queries. Having that limitation in mind, Catalano and Fiore [CF13] presented a simple construction, whose security is based on a stronger model where an adversary can ask for multiple verification queries. For their construction to be simple and efficient, they had to reduce the range of accepted computations.

The main drawback of the CF construction, besides the limited range of computations supported, is that the verification algorithm runs in time proportional to the size of the function. To overcome this problem, a construction with efficient (amortized) verification based on CF was proposed by Backes, Fiore, and Reischuk [BFR13]. The security of their construction is also based on a stronger model, but in order to achieve faster verification times, they introduced an efficient PRF that allows their construction to verify efficiently in an amortized sense.

## 2.3 Succinct Non-Interactive Arguments of Knowledge

It would be possible to construct fully homomorphic signatures by using succinct non-interactive arguments of knowledge (SNARKs) [BCCT11]. Informally, this primitive allows us to create a succinct argument $\pi$ for any NP statement, to prove knowledge of the corresponding witness. The length of $\pi$ is independent of the statement/witness size, and the complexity of verifying $\pi$ only depends on the size of the statement.

Basically, using SNARKs, the output $y$ of a program $f$ can be authenticated by creating a short argument $\pi$ that proves the knowledge of "the input data $D$ along with valid signatures authenticating $D$, such that $f(D) = y$". Because this is an argument of knowledge, if we were able to forge a signature for the output of some program $f$, we would be able to extract a forged signature for the input data $D$, and therefore break the security of signatures.

The main drawback of SNARKs is that they are not efficient in practice, and the current constructions rely on the random oracle model or other non-standard, non-falsifiable assumptions.

## 2.4 Verifiable Computation

Verifiable computation (VC) allows a client with much smaller computational power to outsource a computationally (heavy) task to a remote server while being able to verify the result in a very efficient way. A client sends to the server both the function $f$ and its input $x$, and the server returns $y = f(x)$ as well as a proof that the function was correctly computed over the input $x$. The verification of the proof should be require substantially less computational effort than executing $f$.

In the most commonly used definition of VC proposed by Gennaro, Gentry, and Parno [GGP09], a VC scheme is commonly composed of four algorithms: KeyGen, ProbGen, Compute and Verify. The program $f$ is a parameter of KeyGen, so it outputs a key pair which depends on $f$. The problem generation algorithm ProbGen is executed by the client and encodes the input $x$ of $f$ as a public value $\sigma_x$, which is given to Compute, and as private value used for Verify. Given the public encoding $\sigma_x$, Compute returns an encoding of the output of $f(x)$. To Verify, we simply need the encoding output of $f(x)$ and the private value of ProbGen, and then either false or the actual output of the function is returned. Such a scheme must be correct, secure and efficient.

The correctness property of a VC scheme states that a VC scheme is correct if ProbGen produces values that allows an honest server to compute values that will verify successfully and corresponding to the evaluation of $f$ on those inputs.

Intuitively, a VC scheme is secure if a malicious server cannot trick the verification algorithm to accept an incorrect output. Particularly, for a given function $f$ and input $x$, a malicious server should not be able to convince Verify to output $y'$ such that $f(x) \neq y'$.

Finally, to be efficient means that the time to encode the input and to verify the output must be smaller than the time to compute $f$.

As we can see, VC is very similar to homomorphic schemes. In the case that we are more interested, homomorphic authentication, the server only knows $f$ and the output of $f$, while in VC, the server knows $f$, the input and output $f$. It is actually possible to build a VC scheme with homomorphic signatures or MACs.

Parno et al. [PGHR13] presented a practical VC scheme (Pinocchio), along with an implementation, where a client can efficiently verify the correctness of an outsourced computation. The verification times are in the orders of 10ms and the proof's size is only 288 bytes, with 128 bits of security.

One other work is the one by Ben-Sasson et al. [BCGTV13] which also includes an implementation: TinyRAM, a random-access machine tailored for efficient verification of non-deterministic computations. The verification times are in the orders of 5ms and the proof size is also 288 bytes, with 128 bits of security. The main improvement of TinyRAM over Pinocchio is regarding the Compute algorithm, which they improve about 5.3 times.

## 2.5 Representation of programs

A natural question we have not addressed until now is how can we represent the programs so that both the client and the server can act on them? The common way of doing it is to first convert the original program (in some high level language) to a circuit representation, and then use that as the program description.

A circuit can be either boolean – only boolean operations such as AND or OR allowed – or arithmetic – only arithmetic operations such as $+$ or $\times$. Since a circuit is a directed acyclic graph, these operations are contained within the internal nodes of the graph, while the inputs of the program are nodes with in-degree 0. And obviously the function's output is the node or set of nodes with out-degree 0.

For some cryptographic protocols, specially in the VC setting where fast verification is a must, the notion of circuits is usually extended with some specific gates. Sometimes it is useful to add a split gate which is very fast to verify, but for the purpose of our work we will only deal with arithmetic circuits expressed by $+$ and $\times$ gates because we are not interested in achieving efficient verification, but rather efficient computations.

For our work, we decided to use Pinocchio's [PGHR13] toolchain to convert a piece of `C` code into an appropriate circuit representation. Even though Pinocchio's focus is on a VC scheme with support for general computations, it also contains a "`C` to circuit" converter. More specially, it is able to convert a substantial subset of `C` instructions into both boolean and arithmetic circuits.

In the next chapter we define more precisely both types of circuits, and we also show how we used Pinocchio for our circuit generation.

# Chapter 3

# Computational Model based on Circuits

The canonical theoretical representation of a computer is the Turing machine [Tur37]. Very briefly, this representation can handle general computations and is as efficient as modern random access memory computers up to a polynomial factor. But for a homomorphic scenario we must rely on an alternative computational model, specially – boolean or arithmetic – *circuits* that allows us to represent algorithms in a algebraic form.

These primitives can also handle general computations and almost as efficiently as Turing machines. More precisely, if there is a Turing machine program that evaluates a function $f$ in at most $n$ steps, then there is a circuit representing $f$ with size $\mathcal{O}(n \log n)$ [PF79]. Of course that these values depend on the application we are computing. For example, an arithmetic circuit for matrix multiplication adds essentially no overhead, whereas a boolean circuit for integer multiplication is less efficient than executing a single 32-bit machine assembly instruction.

In a circuit evaluation of a function or program, the number of computational steps does not depend on the inputs of the function. Because of this one cannot take advantage of certain optimizations used in today's computing environments. For instance, the running time of a circuit evaluation for an "easy" input is practically the same as the one with an "hard" input.

## 3.1 Boolean circuits

A *boolean circuit* provides a good mathematical model of the circuitry inside modern computers. They basically compute the boolean function of their bit inputs using only boolean operations. Formally it can be defined as follows.

**Definition 3.1.** *A boolean circuit over the set of boolean variables $X = \{x_1, \ldots, x_n\} \in \{0,1\}^n$ is a directed acyclic graph with the following properties, where each node of the graph is referred to as a* gate.

*(i) Gates with in-degree (or* fan-in*) 0 are* inputs*, and are labeled by either a variable from $X$ or by a boolean constant $c \in \{0, 1\}$.*

*(ii) Gates with fan-in $k > 0$ are labeled by one of the boolean functions* AND, OR, NOT *on $k$ inputs. In the case of* NOT*, $k = 1$.*

*(iii) Gates with out-degree (or* fan-out*) 0 are* outputs*.*

*The* size *of a boolean circuit is the total number of gates. The* depth *is the maximum distance from an input to an output (i.e., the longest directed path in the graph).*

Usually, boolean circuits are composed of only one output gate, and in such cases, it perfectly represents a boolean function $f\colon \{0,1\}^n \to \{0,1\}$.

## 3.2 Arithmetic circuits

Many computational problems can be written down naturally as multivariate polynomials in the input variables. The *arithmetic circuit* is a mathematical model that captures a natural class of algorithms that only use algebraic operations of the underlying field – addition and multiplication – to compute a function. Formally it is defined as follows.

**Definition 3.2** ([SY10, Definition 1.1])**.** *An arithmetic circuit $f$ over the field $\mathbb{F}$ and the set of variables $X = \{x_1, \ldots, x_n\} \in \mathbb{F}^n$ is a directed acyclic graph with the following properties, where each node of the graph is referred to as a* gate*.*

*(i) Gates with in-degree 0 are* input gates*, and are labeled by either a variable from $X$ or a constant field element $c \in \mathbb{F}$.*

*(ii) Every other gate is labeled by either $\times$ (*product gate*) or $+$ (*sum gate*), and have in-degree $k = 2$.*

*(iii) Gates with out-degree 0 are* output gates*.*

*An arithmetic circuit is called a* formula *if it is a directed tree whose edges are directed from the leaves to the root. The* size *of the circuit is the total number of gates, and the* depth *is the maximum distance from an input gate to an output gate.*

For our work, we are specially interested in circuits with just one output gate, which represents the single output of a function that we wish to authenticate in the setting of homomorphic authentication. Since each input gate can only take arbitrary values in $\mathbb{F}$ or some constant $c \in \mathbb{F}$, it is easy to see that the output of each gate has a nice mathematical interpretation: it is simply a multivariate

polynomial (evaluated at the inputs). Arithmetic circuits provide a very compact and versatile way of representing computations that can be expressed as multivariate polynomials.

The polynomial $f \in \mathbb{F}[x_1, \ldots, x_n]$ is computed by an arithmetic circuit as follows. Input gates compute the polynomial defined by their label. Sum gates compute the polynomial obtained by the sum of the two polynomials on their incoming wires. Product gates compute the product of the two polynomials on their incoming wires. The output of the circuit is the polynomial contained on the outgoing wire of the output gate. The *degree of a gate* is defined as the total degree of the polynomial computed by that gates. The *degree of the circuit* is defined as the maximal degree of the gates in the circuit.

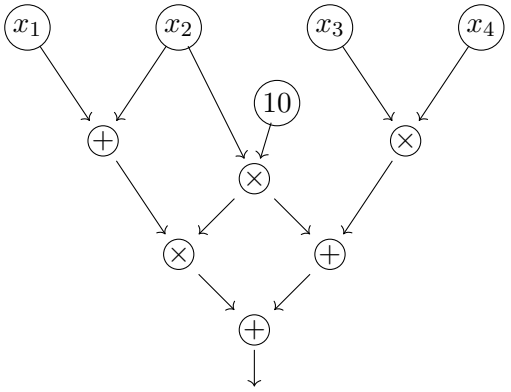



**Figure 3.1:** Representation of the polynomial $f = 10x_2(x_1 + x_2) + ((x_3x_4) + 10x_2)$ as an arithmetic circuit.

Boolean circuits are not as highly structured as arithmetic circuits due to the algebraic nature of of the latter. That is why is is possible to prove results in the arithmetic world that are still considered open in the Boolean world. One very common problem related to circuits is to determine lower bounds i.e., prove that for a circuit computing $f$, there isn't any other better circuit. For arithmetic circuits, there are super-linear bounds known on the size of the circuit, while in the Boolean case, this is much harder to prove.

It is worth noting that when using arithmetic circuits we are usually concerned with polynomials of a particular form, which then fixes the set of computable functions (i.e., polynomials in $\mathbb{F}[X]$). With Boolean circuits it is usually the reverse, where we are interested in computing functions from $\mathbb{F}^{|X|}$ to $\mathbb{F}$. In the arithmetic case, this is because a function may be expressed by a polynomial in several ways, while, in general, a polynomial defines a unique function. A simple example is the function $f = x^2 + x$, which defines the zero polynomial, but only under $\mathbb{F}_2$.

For arithmetic circuits, a natural question that may arise is: since there are only $+$ and $\times$ gates, why not add a $\div$ gate to the set of allowed operations? In the past this model was actually considered, but if we add a $\div$ gate, this gate now computes a rational function instead of a polynomial. Also the

problem of dividing by a zero polynomial arises, so there needs to be a restriction for this case. But as it turns out, it is possible to represent the division gate using only $+$ and $\times$ gates, with some polynomially additional cost. These solutions were first introduced by Strassen, and later by Hrubeš and Yehudayof (see [SY10, Section 2.5] for details on both solutions and proof sketches).

From now on we will only focus on arithmetic circuits, and so when using the term circuit we are referring to an arithmetic circuit.

## 3.3 Circuit Evaluation

We already know how circuits are represented and what operations they compute. But how do we efficiently evaluate them? This is critical for us because the evaluation of a circuit shouldn't become too slow when comparing to a native evaluation (i.e., evaluation a function on a Turing machine model). This is even more critical as circuits start to grow, reaching sizes in the orders of hundreds of inputs and thousands of gates.

If we look at the circuit representation, it is simply a DAG that represents the dependencies between gates. That is, the inputs of a gate $g$ are its dependencies, and therefore we cannot evaluate $g$ before we evaluate its inputs. So we can treat a circuit as a *dependency graph*. For such graphs it is possible to generate an *evaluation order* using some topological sort algorithm. Some of these algorithms perform checks to detect whether the graph is cyclic or not, but for our case we always assume that graphs are acyclic. These kind of algorithms are fundamental for many of common computing applications such as program (re)compilation, system package management or spreadsheet calculators.

The common algorithm used for topological sort is based on the DFS [Tar72], a depth search algorithm over graphs. Basically, edges are explored out of the most recently visited vertex $g$ that still has unexplored edges leaving it. When all of $g$'s edges have been visited, the search "backtracks" to explore edges leaving the vertex from which $g$ was explored. This process continues until all the vertices that are reachable from the original source have been visited. If a vertex is left unvisited, then it is used as a new source for the DFS and the "backtracking" process is repeated again. This process is repeated until all vertices in the graph have been visited. The running time of DFS is $\Theta(V + E)$ with $V$ the number of vertices and $E$ the number of edges. A simple DFS can be implemented as shown in Algorithm 1.

By the end of DFS there are two timestamps in each vertex $u$: $u.d$ represents the time at which it was first visited and $u.f$ represents the time at which all of its children edges where visited. For each vertex $u$, it holds that $u.d < u.f$ and their values are in $\{1, \ldots, 2|V|\}$.

**Algorithm 1** Basic DFS.

```
1: function DFS(G)                                   ▷ Acyclic graph G = (V, E)
2:     for each vertex u ∈ V do
3:         u.color ← WHITE
4:     end for
5:     time ← 0                                      ▷ Global variable to keep track of time
6:     for each vertex u ∈ V do
7:         if u.color == WHITE then
8:             DFS-Visit(G, u)
9:         end if
10:    end for
11: end function

12: function DFS-Visit(G, u)
13:    time ← time + 1
14:    u.d ← time                                    ▷ Time at which white vertex u was discovered
15:    u.color ← GRAY
16:    for each vertex v ∈ G.Adj[u] do
17:        if v.color == WHITE then
18:            DFS-Visit(G, v)
19:        end if
20:    end for
21:    u.color ← BLACK                               ▷ Vertex u is finished
22:    time ← time + 1
23:    u.f ← time                                    ▷ Time at which black vertex u was finished
24: end function
```

A topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering. Any DAG has at least one topological ordering. Given an arithmetic circuit, its topological sort can be viewed as an ordering of its gates along a horizontal line so that all directed edges go from left to right.

With DFS, a topological sorting becomes as simple as the following:

**Algorithm 2** A simple topological sorting algorithm, using DFS as in algorithm 1.

```
1: function Topo-Sort(G)
2:     DFS(G)                  ▷ To compute the finishing times u.f for each vertex of G
3:     Output a list L in order of decreasing finishing times
4: end function
```

Since DFS takes time $\Theta(V + E)$ and the insertions on the list $L$ take time $\mathcal{O}(1)$, the total running time of a topological sorting based on a DFS is $\Theta(V + E)$.

Finally, to perform the circuit evaluation we simply evaluate each gate in the order defined by the list $L$ obtained with Topo-Sort. Each gate evaluation result is kept on a list, where the last index corresponds to the circuit's output gate.

We will be using Pinocchio to generate circuits, and in that case, there is practically no advantage in using a topological sorting algorithm to perform the evaluation because most of the the circuits from Pinocchio are already properly sorted. However, if using custom designed circuits or circuits generated by some other means, it is necessary to generate a topological ordering to be able to efficiently evaluate the circuit.

The reason why we are solely interested in arithmetic circuits is because they allows to represent computations as multivariate polynomials that can be efficiently computed. This is very important specially for our homomorphic scenario.

## 3.4 Circuit generation

Even though arithmetic circuits are quite clean and simple in theory, there are not many tools to easily compile a piece of code into a circuit representation. At this moment, the ones from Parno et al. [PGHR13] (Pinocchio) and from Ben-Sasson et al. [BCGTV13] (TinyRAM) are the most efficient.

Ben-Sasson et al. [BCGTV13] functionality is clearly superior because it is essentially a port of the gcc[1]compiler. Basically, TinyRAM takes a `C` program and compiles it to a specific TinyRAM assembly language, which is equivalent to an arithmetic circuit. Unfortunately, we cannot easily access the arithmetic circuit but only its binary representation.

On the other hand, Pinocchio comes equipped with a compiler that converts a subset of the `C` language and outputs an arithmetic circuit representation, and this is the reason why we are using it instead of TinyRAM. They have support for both boolean and arithmetic circuit generation, but as it was previously mentioned, we are only interested on arithmetic circuits. They also mention that the use of boolean circuits as opposed to arithmetic circuits proved to be much slower, specially on their setting of verifiable computation.

The compiler, written in Python, can process a substantial subset of `C`: global, function and block-scoped variables; arrays, struct's and pointers; function calls, conditional loops; static initializers; arithmetic and bit-wise operators; and pre-processor directives. The piece code that we wish to convert to a circuit must be inside the function `void outsource(struct Input *, struct Output *)`. The parameters describe the input and output values, respectively.

Because circuits only supports expressions and not mutable state and iteration, the `C` program's semantics is restricted. There is no support for dynamic operations (like memory allocations), and everything must be compile-time constant (as pointers or array dereferences).

[1] `https://gcc.gnu.org/`

Their notion of arithmetic circuits is complemented with one more gate operation: a *split gate*. Basically, given a value $a \in \mathbb{F}_p$ where $p$ is an integer with $k$ bits, the split gate outputs to $k$ gates, where each of them represents the bit value $a_i$. Given such binary values, it is possible to compute Boolean functions using only arithmetic gates: NAND$(a, b) = 1 - ab$, AND$(a, b) = ab$, OR$(a, b) = 1 - (1 - a)(1 - b)$; each of these operations comes at the cost of only one multiplication.

The recombination of the $k$ bits into a single gate is given by the expression $\sum_{i=1}^{k} 2^{i-1} a_i$. As they noticed, recombining these $k$ bits into an integer is not very costly as they only perform additions and multiplications by constants, which are practically free.

In the setting of VC and SNARKs[2], the interest is in the verification of computations rather than efficiently compute them homomorphically (which is our case). That is why they introduced a split gate, which is very efficiently verifiable and allows them to compute more operations.

## 3.5 Limitations

One problem with circuit generation has to do with *data dependencies*. While the circuit's inputs do not affect its topology, they affect the program flow and memory accesses. Thus, a circuit must be ready to support a wide range of program flows and memory accesses, despite the fact that its topology has already been set. The technique Pinocchio uses to generate circuits is *program analysis*. With program analysis, if the program and its inputs are known in advance, generating the corresponding circuit is simple: build the circuit's topology to match the pre-determined program flow and its memory accesses. But if only the program is known in advance, and not its inputs, the program must be analyzed piece by piece (i.e., unroll loops, branches, etc) so that the circuit can handle different input values. However, this technique has one big limitation. The class of supported programs is not very broad, as Pinocchio requires arrays accesses and loop iterations bounds to be compile-time constants, which forces us to "write around" this limited functionality.

Unfortunately, the introduction of the split gate is a limitation for us because we are not solely interested in an efficient verification (as with VC and SNARKs), which it limits even more the subset of `C` code that we can convert to arithmetic circuits. This means that we cannot have conditional dependencies which means that everything must be known at compile time (except for the input values), and we cannot also use any bit-wise operation since they give origin to split gates.

However, we can still write simple `C` programs and use Pinocchio to convert them to arithmetic circuits, so that they can be used by the CF scheme.

[2]Pinocchio and TinyRAM were constructed with VC and SNARKs in mind.

# Chapter 4

# Homomorphic Authenticators

A trivial solution to the problem of enforcing authenticity of outsourced computations would be to make the server send the entire data and the computed signatures/MACs (we will refer to both of them as tags from now on) back to the client, and then she would compute the function herself and compare the results. But by using homomorphic authentication primitives, and by sending only the tags and the output of the function back to the client, beyond saving bandwidth, the amount of information revealed to the client about the data set is much smaller. Also, this trivial solution does not guarantee efficiency which is one of the requirements to enforce the authenticity of computations.

Let us now introduce some basic notions and definitions necessary to build homomorphic authentication schemes, as well as some constructions. From now on, it is always assumed that Alice stores the data and tags on some possibly untrusted server.

## 4.1 Homomorphic Signatures

A homomorphic signature scheme allows the computation of functions on previously signed data to obtain a derived signature, and later anyone with the public key can verify the results. This is known as *public verifiability*. The computed signature authenticates both the function and the result of applying the function to the data.

In the only construction presented so far (beyond linear), Boneh and Freeman [BF11] complemented the definition of homomorphic signatures first introduced by Johnson et al. [JMSW02]. Basically, Alice has a (numerical) data set $m_1, \ldots, m_n$, and she signs each message $m_i$, but before signing she augments it with a label $\tau$ and an index $i$. For each message, she signs $(\tau, m_i, i)$ and obtains $n$ independent signatures $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$. The value of the label $\tau$ serves as a name to the data set and binds its members together.

Later, one can ask the server to compute functions on some portion of the data. To compute a function $f$, the server uses a public key homomorphic evaluation algorithm that basically signs the triple $(\tau, m := f(m_1, \ldots, m_n), \langle f \rangle)$, where $\langle f \rangle$ is an encoding of the function $f$. This evaluation algorithm only acts on signatures, not in the original messages. The pair $(m, \sigma)$ can be made public and anyone can check that the server correctly computed $f$ over the data set by verifying that $\sigma$ is a signature on triple $(\tau, m, \langle f \rangle)$. The derived signature $\sigma$ authenticates both the function $f$ and the result of computing $f$ over the data. Using the pair $(m, \sigma)$, one can derive signatures on functions of $m$ and other signed data. The short label $\tau$ is used to prevent mixing of data from different data sets when evaluating functions.

Like "regular" signature schemes, a homomorphic signature scheme consists of the usual algorithms KeyGen, Sign and Vrfy as well as an additional Eval that evaluates a given function on a set of previously signed messages. If $\vec{\sigma}$ is a valid set of signatures authenticating messages $\vec{m}$, then $\mathsf{Eval}(f, \vec{\sigma})$ should be a valid signature for $f(\vec{m})$. It is worth noting once again that Eval only acts on already signed messages.

**Definition 4.1.** *A* homomorphic signature scheme HSIG *is a tuple of PPT algorithms (*KeyGen*,* Sign*,* Vrfy*,* Eval*) as follows:*

$\mathsf{KeyGen}(1^\lambda, n) \to (\mathsf{pk}, \mathsf{sk})$**:** *Takes a security parameter $\lambda$ and a maximum data set size $n$. Outputs a public and secret key,* pk *and* sk*, respectively. The public key also defines a message space $\mathcal{M}$, a signature space $\Sigma$ and a set $\mathcal{F}$ of all "admissible" functions $f\colon \mathcal{M}^n \to \mathcal{M}$.*

$\mathsf{Sign}_{\mathsf{sk}}(\tau, m, i) \to \sigma$**:** *Receives a label $\tau \in \{0,1\}^\lambda$, a message $m \in \mathcal{M}$ and an index $i \in \{1, \ldots, n\}$, and outputs a signature $\sigma \in \Sigma$.*

$\mathsf{Vrfy}_{\mathsf{pk}}(\tau, m, \sigma, f) \to \{\mathsf{accept}, \mathsf{reject}\}$**:** *Receives a label $\tau \in \{0,1\}^\lambda$, a message $m \in \mathcal{M}$, a signature $\sigma \in \Sigma$, a function $f \in \mathcal{F}$, and outputs either* accept *or* reject*.*

$\mathsf{Eval}_{\mathsf{pk}}(\tau, f, \vec{\sigma}) \to \sigma'$**:** *Receives a label $\tau \in \{0,1\}^\lambda$, a function $f \in \mathcal{F}$, a tuple of signatures $\vec{\sigma} \in \Sigma^n$, and outputs a signature $\sigma' \in \Sigma$.*

A signature scheme like the one presented above is said to be *$\mathcal{F}$-homomorphic*, or *homomorphic with respect to $\mathcal{F}$*.

Let $\pi_i\colon \mathcal{M}^n \to \mathcal{M}$ be the function that projects onto the $i$th element, i.e., $\pi_i(m_1, \ldots, m_n) = m_i$. It is required that for all pk generated by KeyGen, $\pi_1, \ldots, \pi_n \in \mathcal{F}$. A homomorphic signature scheme must achieve the following properties: authentication and evaluation correctness, unforgeability and length efficiency.

**Authentication correctness** For each $(\mathsf{pk}, \mathsf{sk})$, all labels $\tau \in \{0,1\}^\lambda$, all $m \in \mathcal{M}$, and all $i \in \{1, \ldots, n\}$, if $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(\tau, m, i)$, then it holds:

$$\Pr[\mathsf{Vrfy}_{\mathsf{pk}}(\tau, m, \sigma, \pi_i) = \mathsf{accept}] \geq 1 - \mathrm{negl}(\lambda). \tag{4.1}$$

**Evaluation correctness** For each $(\mathsf{pk}, \mathsf{sk})$, all $\tau \in \{0,1\}^\lambda$, all tuples $\vec{m} = (m_1, \ldots, m_n) \in \mathcal{M}^n$, and all functions $f \in \mathcal{F}$, if $\sigma_i \leftarrow \mathsf{Sign}_{\mathsf{sk}}(\tau, m_i, i)$ for $i = 1, \ldots, n$ and $\sigma' = \mathsf{Eval}_{\mathsf{pk}}(\tau, f, (\sigma_1, \ldots, \sigma_n))$ it must hold:

$$\Pr[\mathsf{Vrfy}_{\mathsf{pk}}(\tau, f(\vec{m}), \sigma', f) = \mathsf{accept}] \geq 1 - \mathrm{negl}(\lambda). \tag{4.2}$$

The Eval algorithm can take as input derived signatures produced by Eval itself, but doing so for a large number of iterations may eventually reach a point where the input signatures are valid, but the output signature is not. For this reason, and to simplify, the evaluation correctness property is limited so that it only requires that Eval produces a valid output when given as input only signatures produced by the Sign algorithm.

**Unforgeability** Informally, a forgery under a chosen message attack is a valid signature $\sigma$ on a triple $(\tau, m, f)$ such that $m \neq f(m_1, \ldots, m_n)$ where $m_1, \ldots, m_n$ is the data set signed using label $\tau$.

**Definition 4.2.** *A homomorphic signature scheme* $\mathsf{HSIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy}, \mathsf{Eval})$ *is* unforgeable *if for all $n$ the advantage of any PPT adversary $\mathcal{A}$ in the following game is negligible in the security parameter $\lambda$:*

**Setup:** *The challenger obtains* $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KeyGen}(1^\lambda, n)$ *and gives* pk *to $\mathcal{A}$.* pk *defines a message space $\mathcal{M}$, a signature space $\Sigma$, and a set $\mathcal{F}$ of admissible functions $f\colon \mathcal{M}^n \to \mathcal{M}$.*

**Queries:** *Proceeding adaptively, $\mathcal{A}$ specifies a sequence of data sets $\vec{m}_i \in \mathcal{M}^n$. For each $i$, the challenger chooses label $\tau_i$ uniformly from $\{0,1\}^\lambda$ and gives to $\mathcal{A}$ the label $\tau_i$ and the signatures $\sigma_{ij} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(\tau_i, m_{ij}, j)$ for $j = 1, \ldots, n$.*

**Output:** *$\mathcal{A}$ outputs a label $\tau^* \in \{0,1\}^\lambda$, a message $m^* \in \mathcal{M}$, a function $f \in \mathcal{F}$, and a signature $\sigma^* \in \Sigma$.*

The adversary $\mathcal{A}$ wins if $\mathsf{Vrfy}_{\mathsf{pk}}(\tau^*, m^*, \sigma^*, f) = \mathsf{accept}$ and either:

- *Type I forgery*: $\tau^* \neq \tau_i$ for all $i$, or
- *Type II forgery*: $\tau^* = \tau_i$ for some $i$, but $m^* \neq f(\vec{m}_i)$.

The advantage of $\mathcal{A}$ is the probability that $\mathcal{A}$ wins the game.

**Privacy** In this setting, privacy means that given signatures on a data set $\vec{m} \in \mathcal{M}^n$, the derived signatures on messages $f_1(\vec{m}), \ldots, f_s(\vec{m})$ do not leak any information about $\vec{m}$ beyond what is revealed by computing the functions $f_1, \ldots, f_s$ known to the attacker over the data set $\vec{m}$. While the original data set $\vec{m}$ is kept hidden, the fact that a derivation took place is not. The verifier should be able to verify that the correct function was applied to the original data.

More precisely, privacy is ensured using *weakly context hiding*. This means that given signatures on a number of messages derived from two different data sets, the attacker cannot tell from which data set the derived signatures came from, and furthermore that this holds when the secret key is leaked. The reason to be called "weak" is that the original signatures on the data are not public.

**Definition 4.3.** *A homomorphic signature scheme* $\mathsf{HSIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy}, \mathsf{Eval})$ *is* weakly context hiding *if for all* $n$*, the advantage of any PPT adversary* $\mathcal{A}$ *in the following game is negligible in the security parameter* $\lambda$*:*

**Setup:** *The challenger invokes* KeyGen *to generate* $(\mathsf{pk}, \mathsf{sk})$*, and gives the pair to* $\mathcal{A}$*.* pk *defines a message space* $\mathcal{M}$*, a signature space* $\Sigma$*, and a set* $\mathcal{F}$ *of admissible functions* $f \colon \mathcal{M}^n \to \mathcal{M}$

**Challenge:** $\mathcal{A}$ *outputs* $(\vec{m}_0^*, \vec{m}_1^*, f_1, \ldots, f_s)$ *with* $\vec{m}_0^*, \vec{m}_1^* \in \mathcal{M}^n$ *and* $f_1, \ldots, f_s \in \mathcal{F}$*. For all* $i = 1, \ldots, s$ *it satisfies that* $f_i(\vec{m}_0^*) = f_i(\vec{m}_1^*)$*. These functions can be output adaptively after* $\vec{m}_0^*, \vec{m}_1^*$ *are output.*

*Then the challenger generates a random bit* $b \in \{0,1\}$ *and a random label* $\tau \in \{0,1\}^\lambda$*. It signs the messages in* $\vec{m}_b$ *using the label* $\tau$ *to obtain a vector* $\vec{\sigma}$ *of* $n$ *signatures.*

*Next, for* $i = 1, \ldots, s$ *the challenger computes a signature* $\sigma_i = \mathsf{Eval}_{\mathsf{sk}}(\tau, f_i, \vec{\sigma})$*.*

*It sends to* $\mathcal{A}$ *the label* $\tau$ *and the signatures* $\sigma_1, \ldots, \sigma_s$*.*

**Output:** $\mathcal{A}$ *outputs a bit* $b'$*.*

$\mathcal{A}$ *wins the game if* $b = b'$*. The advantage of* $\mathcal{A}$ *is the probability that* $\mathcal{A}$ *wins the game.*

If an attacker wins the previous game, it means that she can determine if the challenge signatures were derived from $\vec{m}_0^*$ or $\vec{m}_1^*$. A signature scheme is $s$*-weakly context hiding* if the attacker cannot win the game after seeing at most $s$ signatures derived from two different data sets.

**Length efficiency** Informally, a signature scheme is length efficient if for a fixed security parameter $\lambda$, the length of the derived signatures depends only logarithmically on the size $n$ of the data set.

**Definition 4.4.** *A homomorphic scheme* $\mathsf{HSIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy}, \mathsf{Eval})$ *is* length efficient *if there is some function* $\mu \colon \mathbb{N} \to \mathbb{R}$ *such that for all* $(\mathsf{pk}, \mathsf{sk})$ *obtained with* KeyGen*, all* $\vec{m} = (m_1, \ldots, m_n) \in \mathcal{M}^n$*,*

*all labels* $\tau \in \{0,1\}^\lambda$*, and all functions* $f \in \mathcal{F}$*, if*

$$\sigma_i \leftarrow \mathsf{Sign}_{\mathsf{sk}}(\tau, m_i, i) \quad \textit{for all } i = 1, \ldots, n \tag{4.3}$$

*then for all* $n > 0$ *and for a derived signature* $\sigma = \mathsf{Eval}_{\mathsf{pk}}(\tau, f, (\sigma_1, \ldots, \sigma_n))$*, it holds:*

$$\Pr[\mathrm{len}(\sigma) \leq \mu(\lambda) \cdot \log n] \geq 1 - \mathrm{negl}(\lambda). \tag{4.4}$$

### 4.1.1 Boneh-Freeman

BF is the first homomorphic signature scheme that is capable of evaluating multivariate polynomials of bounded degree. All previous schemes focused solely on *linear functions*, while their scheme allows the computation of various statistical functions like mean or standard deviation. They also proposed an improved linear scheme with support for linear functions for small fields $\mathbb{F}_p$, specially for $p = 2$.

Here we will present an overview of their two constructions. A more detailed description can be found in [BF11], as well as suitable parameters and sampling algorithms necessary for the full definition of the scheme. Both constructions rely on lattice-based techniques. They are build on the "hash-and-sign" signatures of Gentry, Peikert, and Vaikuntanathan [GPV08].

Abstractly, in GPV, the public key is a lattice $\Lambda \subset \mathbb{Z}^\lambda$ and the secret key is a short basis of $\Lambda$. To sign a message $m$, the holder of the secret key hashes $m$ to an element $H(m) \in \mathbb{Z}^\lambda/\Lambda$ and samples a short vector $\sigma$ from the coset of $\Lambda$ defined by $H(m)$. The verification of $\sigma$ is simply checking that $\sigma$ is short and that $\sigma \bmod \Lambda = H(m)$.

#### Linearly homomorphic scheme

In a (linearly) homomorphic signature scheme, the message space is defined as $\mathbb{F}_p^\lambda$ and a function $f\colon (\mathbb{F}_p^\lambda)^n \to \mathbb{F}_p^\lambda$ is encoded as $f(m_1, \ldots, m_n) = \sum_{i=1}^n c_i m_i$. The $c_i$ are integers in $(-p/2, p/2]$ and the description of $f$ is $\langle f \rangle := (c_1, \ldots, c_n) \in \mathbb{Z}^n$. To authenticate both the message and the function, as well as binding them together, a single GPV signature is computed that is simultaneously a signature on the (unhashed) message $m \in \mathbb{F}_p^\lambda$ and a signature on the hash of $\langle f \rangle$.

Such signature is obtained by using the "intersection method". Boneh and Freeman [BF11] described it as follows. Let $\Lambda_1$ and $\Lambda_2$ be $\lambda$-dimensional integer lattices with $\Lambda_1 + \Lambda_2 = \mathbb{Z}^\lambda$. Suppose $m \in \mathbb{Z}^\lambda/\Lambda_1$ is a message and $\omega_\tau\colon \mathbb{Z}^n \to \mathbb{Z}^\lambda/\Lambda_2$ is a hash function, dependent of $\tau$, that maps encodings of functions $f$ to elements of $\mathbb{Z}^\lambda/\Lambda_2$. Since $m$ defines a coset of $\Lambda_1$ in $\mathbb{Z}^\lambda$ and $\omega_\tau$ defines a coset of $\Lambda_2$ in $\mathbb{Z}^\lambda$, by the CRT the pair $(m, \omega_\tau(\langle f \rangle))$ defines a unique coset of $\Lambda_1 \cap \Lambda_2$ in $\mathbb{Z}^\lambda$. Then, a short

vector $\sigma$ is computed with the property that $\sigma = m \bmod \Lambda_1$ and $\sigma = \omega_\tau(\langle f \rangle) \bmod \Lambda_2$. The vector $\sigma$ is a signature on $(\tau, m, \langle f \rangle)$ that "binds" $m$ and $\omega_\tau(\langle f \rangle)$ – an attacker cannot generate a new short vector $\sigma'$ from $\sigma$ such that $\sigma = \sigma' \bmod \Lambda_1$ but $\sigma \neq \sigma' \bmod \Lambda_2$.

In this linear scheme, $\mathbb{F}_p^\lambda \cong \mathbb{Z}^\lambda/\Lambda_1$ with the isomorphism given explicitly by the map $\vec{x} \mapsto (\vec{x} \bmod \Lambda_1)$, and $\mathbb{F}_q^\ell \cong \mathbb{Z}^\lambda/\Lambda_2$.

Using the above method, $\mathsf{Sign}_{\mathsf{sk}}(\tau, m, i)$ generates a signature on $(\tau, m, \langle \pi_i \rangle)$, where $\pi_i$ is the $i$th *projection function* defined by $\pi_i(m_1, \ldots, m_n) = m_i$ and encoded by $\langle \pi_i \rangle = \vec{e}_i$, the $i$th unit vector in $\mathbb{Z}^n$.

To authenticate the linear combination $m = \sum_{i=1}^n c_i m_i$ previously authenticated, compute the signature $\sigma := \sum_{i=1}^n c_i \sigma_i$. If $n$ and $p$ are sufficiently small, then $\sigma$ is a short vector. The following homomorphic property is achieved:

$$\begin{aligned} \sigma \bmod \Lambda_1 &= \sum_{i=1}^n c_i m_i = m, \text{and} \\ \sigma \bmod \Lambda_2 &= \sum_{i=1}^n c_i \omega_\tau(\langle \pi_i \rangle) = \sum_{i=1}^n c_i \omega_\tau(\vec{e}_i). \end{aligned} \tag{4.5}$$

If $\omega_\tau$ is linearly homomorphic then $\sum_{i=1}^n c_i \omega_\tau(\vec{e}_i) = \omega_\tau((c_1, \ldots, c_n))$ for all $c_i \in \mathbb{Z}$. Since $(c_1, \ldots, c_n)$ is exactly the encoding $\langle f \rangle$, the signature $\sigma$ authenticates both the message $m$ and the fact that $m$ is indeed the output of $f$ applied to the original messages $m_1, \ldots, m_n$.

Verification of a signature $\sigma$ is simply checking that

$$\begin{aligned} \sigma \bmod \Lambda_1 &\stackrel{?}{=} m, \\ \sigma \bmod \Lambda_2 &\stackrel{?}{=} \omega_\tau(\langle f \rangle). \end{aligned} \tag{4.6}$$

**Correctness** The above linearly homomorphic scheme satisfies both authentication and evaluation correctness. The proofs can be found in [BF11].

**Unforgeability** The security is based on the SIS problem on $q$-ary lattices for some prime $q$. More precisely, for security parameter $\lambda$ it follows:

**Theorem 4.5.** *If $\mathsf{SIS}_{q,\lambda,\beta}$ is infeasible for a suitable $\beta$, then the above linearly homomorphic scheme is unforgeable in the random oracle model.*

To prove it, it is shown that a successful forger can be used to solve the SIS problem in the lattice $\Lambda_2$, which for random $q$-ary lattices (and suitable parameters) is as hard as standard worst case lattice problems. The full proof and the suitable parameters can be found in [BF11].

**Privacy** The above linearly homomorphic signature scheme is weakly context hiding. Based on the fact that the distribution of a linear combination of samples from a discrete Gaussian is itself a discrete Gaussian, it is proved that a derived signature on a linear combination $m' = \sum_{i=1}^{n} c_i m_i$ depends (up to negligible distance) only on $m'$ and the $c_i$, and not on the original messages $m_i$. The full proof can be found in [BF11].

**Length efficiency** The bit length of a derived signature only depends logarithmically on the size of the data set. The proof, along with the exact length of a derived signature, can be found in [BF11].

### Polynomially homomorphic scheme

The basic idea for this scheme is as follows: what if $\mathbb{Z}^\lambda$ has a ring structure and lattices $\Lambda_1, \Lambda_2$ are ideals? Then the maps $\vec{x} \mapsto \vec{x} \bmod \Lambda_i$ are ring homomorphisms, and therefore adding or multiplying signatures corresponds to adding or multiplying the corresponding messages or functions. Since any polynomial can be computed by repeated additions and multiplications, by adding this structure to the previous scheme it is possible to authenticate polynomial functions on messages.

Let $F(x) \in \mathbb{Z}[x]$ be a monic, irreducible polynomial of degree $\lambda$ and let $R$ be the ring $\mathbb{Z}[x]/(F(x))$. $R$ is isomorphic to $\mathbb{Z}^\lambda$ and ideals in $R$ correspond to integer lattices in $\mathbb{Z}^\lambda$ under the "coefficient embedding". This embedding maps a polynomial in $R$ to an element in an ideal lattice as follows: if $g(x) = g_0 + g_1 x + \cdots + g_{\lambda-1} x^{\lambda-1}$, then coefficient embedding can be defined as: $g(x) \mapsto (g_0, \ldots, g_{\lambda-1}) \in \mathbb{Z}^\lambda$.

$\Lambda_1$ and $\Lambda_2$ are specially chosen to be degree one prime ideals $\mathfrak{p}, \mathfrak{q} \subset R$ of norm $p, q$ respectively, with $\mathfrak{p} = (p, x - a), \mathfrak{q} = (q, x - b)$. $R/\mathfrak{p}$ and $R/\mathfrak{q}$ are isomorphic with $\mathbb{F}_p$ and $\mathbb{F}_q$, respectively. The message space is defined by $\mathbb{F}_p$. Sign is now exactly as in the linearly homomorphic scheme, and it returns a short signature $\sigma \in R$.

When considering polynomial functions on $\mathbb{F}_p[x_1, \ldots, x_n]$, the projection functions $\pi_i$ are exactly the linear monomials $x_i$, and any polynomial function can be obtained by adding and multiplying monomials. If an ordering on all monomials of the form $x_1^{e_1} \cdots x_n^{e_n}$ is fixed, then it is possible to encode any polynomial function as its vector of coefficients, with the $i$-th unit vector $\vec{e}_i$ representing the linear monomials $x_i$ for $i = 1, \ldots, n$.

The hash function $\omega_\tau$ is defined exactly as in the linear scheme: for a function $f$ in $\mathbb{F}_p[x_1, \ldots, x_n]$ encoded as $\langle f \rangle = (c_1, \ldots, c_\ell) \in \mathbb{Z}^\ell$, define a polynomial $\hat{f} \in \mathbb{Z}[x_1, \ldots, x_n]$ that reduces to $f$ mod $p$. Then, $\omega_\tau(\langle f \rangle) = \hat{f}(\alpha_1, \ldots, \alpha_n)$ where each $\alpha_i \in \mathbb{F}_q$ is the output of the hash function $H(\tau || i)$.

The evaluation algorithm uses the lifting of $f$ to $\hat{f}$. Given $f$ and the signatures $\sigma_1, \ldots, \sigma_n \in R$ on messages $m_1, \ldots, m_n \in \mathbb{F}_p$, the signature on $f(m_1, \ldots, m_n)$ is given by $\hat{f}(\sigma_1, \ldots, \sigma_n)$. It returns a short signature $\sigma \in R$ on triple $(\tau, m, \langle f \rangle)$ such that $\sigma = m \bmod \mathfrak{p}$ and $\sigma = \omega_\tau(\langle f \rangle) \bmod \mathfrak{q}$. Let $\sigma_i$

be a signature on $(\tau, m_i, \langle f_i \rangle)$ for $i = 1, 2$. The following homomorphic properties can be observed:

$$\begin{aligned} \sigma_1 + \sigma_2 \text{ is a signature on } (\tau, m_1 + m_2, \langle f_1 + f_2 \rangle) \\ \sigma_1 \cdot \sigma_2 \text{ is a signature on } (\tau, m_1 \cdot m_2, \langle f_1 \cdot f_2 \rangle) \end{aligned} \tag{4.7}$$

The verification is essentially the same as in the linearly homomorphic scheme.

**Correctness** The above scheme satisfies both authentication and evaluation correctness properties. Proof can be found in [BF11].

**Unforgeability** To prove that the above scheme is unforgeable, it is shown that a successful forger can be used to find a solution to the SIVP in the lattice $\Lambda_2$, which in this scheme is the ideal $\mathfrak{q}$.

**Privacy** For large $p$, this homomorphic scheme is not weakly context hiding as defined in Definition 4.3. In [BF11] is is shown exactly why the signature $\sigma$ leaks information about the original messages $m_0, m_1$.

**Length efficiency** The bit length of a derived signature only depends logarithmically on the size of the data set, for polynomials of bounded constant degree and for small coefficients. The proof, along with the exact length of a derived signature, can be found in [BF11].

## 4.2 Homomorphic MACs

The symmetric analogue of homomorphic signatures are homomorphic MACs. Basically, Alice has a secret key which is used to generate a tag $\sigma$ that authenticates a message $m$ under label $\tau$. Given a labeled program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ and a set of tags $\sigma_1, \ldots, \sigma_n$, anyone can execute the homomorphic evaluation algorithm over $\mathcal{P}(\sigma_1, \ldots, \sigma_n)$ to generate a short tag $\sigma'$ that authenticates $m' = \mathcal{P}(m_1, \ldots, m_n)$ as the output of the $\mathcal{P}$ executed over inputs labeled by $\tau_1, \ldots, \tau_n$, respectively.

There are three main properties that a homomorphic MAC scheme is required to satisfy. (1) It must be *secure*, i.e., an adversary that asks for tags authenticating some messages of his own choice should not be able to produce valid tags authenticating messages that are not obtained as the output of $\mathcal{P}$. (2) It should be *succinct*, i.e., the output of $\mathcal{P}$ can be certified using much less communication that than of sending the original authenticated messages. (3) Lastly, it should be *composable* meaning that tags authenticating previous outputs of $\mathcal{P}$ should be usable as inputs to further authenticate new

computations, i.e., the tag authenticating the output of $\mathcal{P}$ can be used as one of the inputs of another labeled program $\mathcal{P}'$.

However, unlike in homomorphic signatures, it is necessary to explain what it really means to authenticate a message as the output of a labeled program. In a scenario where many users share the secret key and authenticate various data-items without keeping any local or joint state, it is necessary to specify which data is being authenticated and which data the program $\mathcal{P}$ should be evaluated on. With this in mind, Gennaro and Wichs [GW12] defined the concept of *labeled programs*[1].

**Labeled Programs** *Labels* are used to index some data. A labeled program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ consists of a circuit[2] $f\colon \mathbb{F}^n \to \mathbb{F}$ and a distinct input label $\tau_i \in \{0,1\}^\lambda$ for each input wire $i \in \{1, \ldots, n\}$ of the circuit. This can be seen as way to give useful names to the variables of a program, without knowing the input values – as an example, we can consider a program that outputs the average salaries of a company, and each $\tau_i$ can be of the form "salary of worker $i$". A message $m$ is then authenticated with respect to a label $\tau$, and the value of the label does not need to have any meaningful semantics. This label-message association basically means that the value $m$ can be assigned to those input variables of a labeled program $\mathcal{P}$ whose label is $\tau$. However, with this definition, a label *cannot* be re-used for multiple messages, i.e., two distinct messages $m, m'$ cannot be authenticated with respect to the same label $\tau$.

Given some labeled programs $\mathcal{P}_1, \ldots, \mathcal{P}_t$ and a circuit $g\colon \mathbb{F}^t \to \mathbb{F}$, the *composed program* $\mathcal{P}^* = g(\mathcal{P}_1, \ldots, \mathcal{P}_t)$ consists in evaluating a circuit $g$ on the outputs of $\mathcal{P}_1, \ldots, \mathcal{P}_t$. The labeled inputs of the composed program $\mathcal{P}^*$ are just all the *distinct* labeled inputs of $\mathcal{P}_1, \ldots, \mathcal{P}_t$, and all the input wires with the same label are put together in a single input wire. The *identity program* with label $\tau$ is denoted by $\mathcal{I}_\tau := (g_{\text{id}}, \tau)$, where $g_{\text{id}}$ is the *canonical identity circuit* and $\tau \in \{0,1\}^\lambda$ is some input label. Any program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ can be written as a composition of programs $\mathcal{P} = f(\mathcal{I}_{\tau_1}, \ldots, \mathcal{I}_{\tau_n})$.

Given a labeled program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ and a set of tags $\sigma_1, \ldots, \sigma_n$ that authenticates messages $m_i$ under label $\tau_i$ , anyone can run the homomorphic evaluation algorithm that takes an input the tuple $(\mathcal{P}, \sigma_1, \ldots, \sigma_n)$ and whose output $\sigma'$ will authenticate $m'$ as the output of $\mathcal{P}(m_1, \ldots, m_n)$[3]. Then the secret-key verification algorithm takes as input the triple $(m', \mathcal{P}, \sigma')$ and verifies that $m'$ is indeed the output of the program $\mathcal{P}$ executed over some previously authenticated and labeled data, without knowing the value of the original data.

[1] Gennaro and Wichs defined labeled programs for boolean circuits $f\colon \{0,1\}^n \to \{0,1\}$, but here we will consider the case for arithmetic circuits $f\colon \mathbb{F}^n \to \mathbb{F}$ where $\mathbb{F}$ is some finite field, like $\mathbb{Z}_p$ for some prime $p$.

[2] Or function.

[3] To be more precise, $\sigma'$ only certifies $m'$ as the output of a specific program $\mathcal{P}$.

**Definition 4.6.** *A* homomorphic message authenticator scheme $\mathsf{HMAC}$ *is a tuple of PPT algorithms (*$\mathsf{KeyGen}$*,* $\mathsf{Auth}$*,* $\mathsf{Vrfy}$*,* $\mathsf{Eval}$*), where* $\mathcal{M}$ *defines a message space, as follows:*

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{ek}, \mathsf{sk})$**:** *Takes a security parameter* $\lambda$*. Outputs a public evaluation key* $\mathsf{ek}$ *and a secret key* $\mathsf{sk}$*.*

$\mathsf{Auth}_{\mathsf{sk}}(\tau, m) \to \sigma$**:** *Receives an input-label* $\tau \in \{0,1\}^\lambda$ *and a message* $m \in \mathcal{M}$*, and outputs a tag* $\sigma$*.*

$\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}$**:** *Receives a message* $m \in \mathcal{M}$*, a labeled program* $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ *and a tag* $\sigma$*, and outputs either* $\mathsf{accept}$ *or* $\mathsf{reject}$*.*

$\mathsf{Eval}_{\mathsf{ek}}(f, \vec{\sigma}) \to \sigma'$**:** *Receives a circuit* $f\colon \mathcal{M}^n \to \mathcal{M}$ *and a vector of tags* $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$ *, and outputs a new tag* $\sigma'$*.*

A homomorphic MAC scheme must achieve the following properties: authentication and evaluation correctness, succinctness and security.

**Authentication correctness** For any $m \in \mathcal{M}$, all keys $(\mathsf{ek}, \mathsf{sk}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$, any label $\tau \in \{0,1\}^\lambda$, and any tag $\sigma \leftarrow_\$ \mathsf{Auth}_{\mathsf{sk}}(\tau, m)$, it holds:

$$\Pr[\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{I}_\tau, \sigma) = \mathsf{accept}] = 1. \tag{4.8}$$

**Evaluation correctness** For a pair $(\mathsf{ek}, \mathsf{sk}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$, a circuit $g\colon \mathcal{M}^t \to \mathcal{M}$ and any set of triples $\{(m_i, \mathcal{P}_i, \sigma_i)\}_{i=1}^t$ such that $\mathsf{Vrfy}_{\mathsf{sk}}(m_i, \mathcal{P}_i, \sigma_i) = \mathsf{accept}$. If $m^* = g(m_1, \ldots, m_t)$, $\mathcal{P}^* = g(\mathcal{P}_1, \ldots, \mathcal{P}_t)$, and $\sigma^* = \mathsf{Eval}_{\mathsf{ek}}(g, (\sigma_1, \ldots, \sigma_t))$, then it must hold:

$$\mathsf{Vrfy}_{\mathsf{sk}}(m^*, \mathcal{P}^*, \sigma^*) = \mathsf{accept}. \tag{4.9}$$

**Succinctness** The size of a computed tag is bounded by some fixed polynomial in the security parameter $\mathsf{poly}(\lambda)$ which is independent of the number $n$ of inputs taken by the evaluated circuit.

**Security** A homomorphic MAC scheme has to satisfy the following notion of unforgeability.

**Definition 4.7.** *A homomorphic MAC scheme* $\mathsf{HMAC} = (\mathsf{KeyGen}, \mathsf{Auth}, \mathsf{Vrfy}, \mathsf{Eval})$ *is* unforgeable *if the advantage of any PPT adversary* $\mathcal{A}$ *in the following game is negligible in the security parameter* $\lambda$*.*

**Setup** *The challenger generates* $(\mathsf{ek}, \mathsf{sk}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$ *and gives the evaluation key* $\mathsf{ek}$ *to* $\mathcal{A}$*. It also initializes a list* $Q = \emptyset$*.*

**Authentication queries** *The adversary can adaptively ask for tags on label-message pairs of her choice. Given a query $(\tau, m)$, if there is some $(\tau, m') \in Q$ for some $m' \neq m$, then the challenger ignores the query. Otherwise, it computes $\sigma \leftarrow \mathsf{Auth}_{\mathsf{sk}}(\tau, m)$, returns $\sigma$ to $\mathcal{A}$ and updates the list $Q = Q \cup (\tau, m)$. If $(\tau, m) \in Q$ (i.e., the query was previously made), then the challenger returns the same tag generated before.*

**Verification queries** *The adversary has access to a verification oracle. $\mathcal{A}$ can submit a query $(m, \mathcal{P}, \sigma)$ and the challenger replies with the output of $\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}, \sigma)$.*

**Forgery** *Eventually, the adversary outputs a forgery $(m^*, \mathcal{P}^* = (f^*, \tau_1^*, \ldots, \tau_n^*), \sigma^*)$. Notice that such tuple can also be returned by $\mathcal{A}$ as a verification query $(m^*, \mathcal{P}^*, \sigma^*)$.*

*Before describing the output of this game, it is necessary to define the notion of a well-defined program with respect to a list $Q$. This notion intends to capture formally, which tuples generated by the adversary $\mathcal{A}$ should be considered as valid forgeries. Because we are dealing with a homomorphic primitive, it should be possible to differentiate genuine tags produced by* Eval *from tags produced in another, possibly malicious, way.*

**Well-defined program** *A labeled program $\mathcal{P}^* = (f^*, \tau_1^*, \ldots, \tau_n^*)$ is* well-defined with respect to $Q$ *if either one of the following conditions is met:*

1. *For some $i \in \{0, \ldots, n\}$, there is a tuple $(\tau_i^*, \cdot) \notin Q$ (i.e., $\mathcal{A}$ never asked authentication queries with label $\tau_i^*$), and $f^*(\{m_j\}_{(\tau_j, m_j) \in Q} \cup \{\tilde{m}_j\}_{(\tau_j, \cdot) \notin Q})$ outputs the same value for all possible values of $\tilde{m}_j \in \mathcal{M}$. This means that the inputs $\tilde{m}_j$ do not affect the behavior of $f^*$* [4].
2. *$Q$ contains tuples $(\tau_i^*, m_i)$ for some messages $m_1, \ldots, m_n$, i.e., the entire input space of $f^*$ has been authenticated.*

*The adversary $\mathcal{A}$ wins if $\mathsf{Vrfy}_{\mathsf{sk}}(m^*, \mathcal{P}^*, \sigma^*) = \mathsf{accept}$ and either:*

- Type I forgery: *$\mathcal{P}^*$ is not well-defined on $Q$ or,*
- Type II forgery: *$\mathcal{P}^*$ is well-defined on $Q$ and $m^* \neq f^*(\{m_j\}_{(\tau_j, m_j) \in Q})$, which means that $m^*$ is not the correct output of the labeled program $\mathcal{P}^*$ when executed on previously authenticated messages $(m_1, \ldots, m_n)$.*

Notice, however, that even maliciously generated tags should not necessarily be considered as forgeries. This is because the adversary $\mathcal{A}$ can trivially modify a circuit $C$ she is allowed to evaluate,

[4] Equivalently, it states that $f^*(\{m_j\}_{(\tau_j, m_j) \in Q} \cup \{\tilde{m}_j\}_{(\tau_j, \cdot) \notin Q})$ is semantically equivalent to $f^*(\{m_j\}_{(\tau_j, m_j) \in Q})$.

by adding dummy gates and inputs that are simply ignored in the evaluation of the modified circuit. This does not constitute an infringement of the security requirements because the notion of a well-defined program $\mathcal{P}$ captures this exact case: either $\mathcal{P}$ is run on legal inputs (i.e., inputs in $Q$) only, or, if this is not the case, those inputs not queried (i.e., the dummy inputs not in $Q$) do not affect the computation in any way.

The previous security definition is very similar to the one from fully homomorphic MACs of Gennaro and Wichs [GW12], except for two modifications. Here it is explicitly allowed to the adversary to query the verification oracle, and forgeries are slightly different. In [GW12], Type I forgeries are defined as ones where at least one new label is present, and Type II forgeries contain only labels that have been queried, but $m^*$ is not the correct output of $\mathcal{P}$ when executed over previously authenticated messages.

For arbitrary computations, there is no efficient way to check whether a program is well-defined with respect to a list $Q$. The main problem is to check the first condition, i.e., whether a program always outputs the same value for all possible choices of inputs that were not queried. However, for the case of arithmetic circuits defined over the finite field $\mathbb{Z}_p$, where $p$ is a prime of roughly $\lambda$ bits, and whose degree is bounded by a polynomial, this check can be efficiently performed. In a follow-up work by Catalano et al. [CFGN14], it is noted that testing whether a program is well-defined can be done for arithmetic circuits of degree $d$, over a finite field $\mathbb{F}$ of order $p$ such that $\frac{d}{p} < \frac{1}{2}$.

As one can observe from the authentication queries phase of the previous game, it is explicitly not allowed to re-use a label to authenticate more than one value. This is basically a way to keep track of the authenticated inputs. This is a restriction that has been present on all previous works on homomorphic authentication primitives.

Backes, Fiore, and Reischuk [BFR13] extended the notion of labeled programs in order to solve the problem of label re-use. Their notion of *multi-labeled programs* allows the partial, but safe, re-use of labels.

**Multi-labeled Programs** A *multi-labeled program* $\mathcal{P}_\Delta$ is a pair $(\mathcal{P}, \Delta)$ where $\mathcal{P} = f(\tau_1, \ldots, \tau_n)$ is a labeled-program and $\Delta \in \{0,1\}^\lambda$ is a binary string denominated *data set label*. Basically, the combination of a input label $\tau_i$ and a data set label $\Delta$, defined as a multi-label $\mathsf{L} = (\Delta, \tau_i)$, is used to uniquely identify a specific data item. In particular, binding a message $m_i$ with a multi-label $\mathsf{L} = (\Delta, \tau_i)$ means that $m_i$ can be assigned to those input variables with input label $\tau_i$. The multi-label $\mathsf{L}$ uniquely identifies the message $m_i$. While the re-use of a multi-label $\mathsf{L}$ is not allowed, the re-use of a input label $\tau_i$ is allowed, instead. Composition of multi-labeled programs within the same data set is possible. Given multi-labeled programs $(\mathcal{P}_1, \Delta), \ldots, (\mathcal{P}_t, \Delta)$ having the same data set label $\Delta$, and given a function

$g\colon \mathcal{M}^t \to \mathcal{M}$, the *composed multi-labeled program* $\mathcal{P}^*_\Delta$ is the pair $(\mathcal{P}^*, \Delta)$ where $\mathcal{P}^*$ is the composed program $g(\mathcal{P}_1, \ldots, \mathcal{P}_t)$, and $\Delta$ is the data set label shared by all $\mathcal{P}_i$. If $f_{\text{id}}\colon \mathcal{M} \to \mathcal{M}$ is the canonical identity function and $\mathsf{L} = (\Delta, \tau) \in (\{0,1\}^\lambda)^2$ is a multi-label, then $\mathcal{I}_\mathsf{L} = (f_{\text{id}}, \mathsf{L})$ is the *identity multi-labeled program* for data set $\Delta$ and input label $\tau$. Like in labeled programs, any multi-labeled program $\mathcal{P}_\Delta = ((f, \tau_1, \ldots, \tau_n), \Delta)$ can be expressed as the composition of $n$ identity multi-labeled programs $\mathcal{P}_\Delta = (\mathcal{I}_{\mathsf{L}_1}, \ldots, \mathcal{I}_{\mathsf{L}_n})$ where $\mathsf{L}_i = (\Delta, \tau_i)$.

Having defined the notion of a multi-labeled program, the definition of a homomorphic message authenticator from Definition 4.6 can be adapted to support multi-labeled programs.

**Definition 4.8.** *A homomorphic message authenticator* HMAC-ML *is a tuple of PPT algorithms (*KeyGen*,* Auth*,* Vrfy*,* Eval*), where $\mathcal{M}$ defines a message space, as follows:*

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{ek}, \mathsf{sk})$**:** *Takes a security parameter $\lambda$. Outputs a public evaluation key* ek *and a secret key* sk*.*

$\mathsf{Auth}_\mathsf{sk}(\mathsf{L}, m) \to \sigma$**:** *Receives a multi-label $\mathsf{L} = (\Delta, \tau) \in (\{0,1\}^\lambda)^2$ and a message $m \in \mathcal{M}$, and outputs a tag $\sigma$.*

$\mathsf{Vrfy}_\mathsf{sk}(m, \mathcal{P}_\Delta, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}$**:** *Receives a message $m \in \mathcal{M}$, a multi-labeled program $\mathcal{P}_\Delta = ((f, \tau_1, \ldots, \tau_n), \Delta)$ and a tag $\sigma$, and outputs either accept or reject.*

$\mathsf{Eval}_\mathsf{ek}(f, \vec{\sigma}) \to \sigma'$**:** *Receives a circuit $f\colon \mathcal{M}^n \to \mathcal{M}$ and a vector of tags $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$, and outputs a new tag $\sigma'$.*

**Authentication correctness** For any message $m \in \mathcal{M}$, all keys $(\mathsf{ek}, \mathsf{sk}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$, any multi-label $\mathsf{L} = (\Delta, \tau) \in (\{0,1\}^\lambda)^2$, and any tag $\sigma' \leftarrow_\$ \mathsf{Auth}_\mathsf{sk}(\mathsf{L}, m)$, it holds:

$$\Pr[\mathsf{Vrfy}_\mathsf{sk}(m, \mathcal{I}_\mathsf{L}, \sigma') = \mathsf{accept}] = 1. \tag{4.10}$$

**Evaluation correctness** For a pair $(\mathsf{ek}, \mathsf{sk}) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$, a circuit $g\colon \mathcal{M}^t \to \mathcal{M}$ and any set of triples $\{(m_i, \mathcal{P}_{\Delta,i}, \sigma_i\}_{i=1}^t$ such that all multi-label programs $\mathcal{P}_{\Delta,i} = (\mathcal{P}_i, \Delta)$ and $\mathsf{Vrfy}_\mathsf{sk}(m_i, \mathcal{P}_{\Delta,i}, \sigma_i) = \mathsf{accept}$. If $m^* = g(m_1, \ldots, m_t)$, $\mathcal{P}^* = g(\mathcal{P}_1, \ldots, \mathcal{P}_t)$, and $\sigma^* = \mathsf{Eval}_\mathsf{ek}(g, (\sigma_1, \ldots, \sigma_t))$, then it must hold:

$$\Pr[\mathsf{Vrfy}_\mathsf{sk}(m^*, \mathcal{P}^*_\Delta, \sigma^*) = \mathsf{accept}] = 1. \tag{4.11}$$

**Succinctness** The size of a computed tag is bounded by some fixed polynomial in the security parameter $\mathsf{poly}(\lambda)$ which is independent of the number $n$ of inputs taken by the evaluated circuit.

**Security** The security property is extended from Catalano and Fiore [CF13] to the model of multi-labeled programs. A homomorphic MAC scheme has to satisfy the following notion of unforgeability.

**Definition 4.9.** *A homomorphic MAC scheme* $\mathsf{HMAC} = (\mathsf{KeyGen}, \mathsf{Auth}, \mathsf{Vrfy}, \mathsf{Eval})$ *is* unforgeable *if the advantage of any PPT adversary* $\mathcal{A}$ *in the following game is negligible in the security parameter* $\lambda$*.*

**Setup** *The challenger generates* $(\mathsf{ek}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KeyGen}(1^{\lambda})$ *and gives the evaluation key to* $\mathcal{A}$*.*

**Authentication queries** *The adversary can adaptively ask for tags on multi-labels and messages of her choice. Given a query* $(\mathsf{L}, m)$ *where* $\mathsf{L} = (\Delta, \tau)$*, if this is the first query with data set* $\Delta$*, the challenger initializes a list* $Q_{\Delta} = \emptyset$*. Then, if* $(\tau, m') \in T_{\Delta}$ *for some* $m' \neq m$*, the challenger ignores the query. Otherwise, if* $(\tau, \cdot) \notin Q_{\Delta}$*, the challenger computes* $\sigma' \leftarrow_{\$} \mathsf{Auth}_{\mathsf{sk}}(\mathsf{L}, m)$*, returns* $\sigma'$ *to* $\mathcal{A}$ *and updates list* $Q_{\Delta} = Q_{\Delta} \cup (\tau, m)$*. If* $(\tau, m) \in Q_{\Delta}$*, then the challenger returns the same tag generated before.*

**Verification queries** *The adversary has access to a verification oracle.* $\mathcal{A}$ *can submit a query* $(m, \mathcal{P}_{\Delta}, \sigma)$*, and the challenger replies with the output of* $\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}_{\Delta}, \sigma)$*.*

**Forgery** *The game ends when* $\mathcal{A}$ *returns a forgery* $(m^*, \mathcal{P}^*_{\Delta^*} = (\mathcal{P}^*, \Delta^*), \sigma^*)$ *for some* $\mathcal{P}^* = (f^*, \tau_1^*, \ldots, \tau_n^*)$*. Notice that such tuple can be returned by* $\mathcal{A}$ *as a verification query* $(m^*, \mathcal{P}^*_{\Delta^*}, \sigma^*)$*.*

*Just as with labeled programs, it is necessary to define the notion of well-defined programs with respect to a list* $Q_{\Delta}$*. The notion is exactly the same as for labeled programs, except that now we are within a data set* $\Delta$*.*

*The adversary* $\mathcal{A}$ *wins the game if* $\mathsf{Vrfy}_{\mathsf{sk}}(m^*, \mathcal{P}^*_{\Delta^*}, \sigma^*) = \mathsf{accept}$ *and one of the following holds:*

- Type I Forgery*: no list* $Q^*_{\Delta}$ *was created, i.e., no message* $m$ *has been authenticated within data set label* $\Delta^*$*.*
- Type II Forgery*:* $\mathcal{P}^*$ *is well-defined with respect to* $Q_{\Delta^*}$ *and* $m^* \neq f^*(\{m_j\}_{(\tau_j, m_j) \in Q_{\Delta^*}})$*, i.e.,* $m^*$ *is not the correct output of labeled program* $\mathcal{P}^*$ *when executed on previously authenticated message* $(m_1, \ldots, m_n)$*.*
- Type III Forgery*:* $\mathcal{P}^*$ *is not well-defined with respect to* $Q_{\Delta^*}$*.*

This definition of security is similar to the one from Catalano and Fiore [CF13] previously presented, but extended to the model of multi-labeled programs. Just as with labeled programs, it is not possible to check in polynomial time if whether an arbitrary computation is well-defined with respect to a list $Q$, but for the specific case of arithmetic circuits defined over the finite field $\mathbb{Z}_p$ and of polynomial degree, this check can be efficiently performed.

**Efficient Verification** The notion of multi-labels by itself does not guarantee an efficient verification algorithm. To achieve this it is necessary to introduce a property of efficient verification. This efficiency property is defined in an amortized sense, meaning that the verification is more efficient when the same program $\mathcal{P}$ is executed on different data sets. To achieve efficient verification, the definition of a homomorphic MAC scheme previously defined has to be augmented with two new algorithms.

**Definition 4.10.** *A homomorphic MAC scheme* $\mathsf{HMAC\text{-}ML} = (\mathsf{KeyGen}, \mathsf{Auth}, \mathsf{Vrfy}, \mathsf{Eval})$ *satisfies* efficient verification *if there exist two additional algorithms (*$\mathsf{VrfyPrep}$, $\mathsf{EffVrfy}$*) as follows:*

$\mathsf{VrfyPrep}_{\mathsf{sk}}(\mathcal{P}) \to \mathsf{vk}_{\mathcal{P}}$**:** *Given a labeled program* $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$*, it generates a* concise *verification key* $\mathsf{vk}_{\mathcal{P}}$*. This verification key does not depend on any data set label* $\Delta$*.*

$\mathsf{EffVrfy}_{\mathsf{sk},\mathsf{vk}_{\mathcal{P}}}(\Delta, m, \sigma)$**:** *Given a data set label* $\Delta$*, a message* $m \in \mathcal{M}$ *and a tag* $\sigma$*, it outputs either accept or reject.*

These new algorithms need to achieve two properties: correctness and amortized efficiency.

**Correctness** Let $(\mathsf{ek}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KeyGen}(1^{\lambda})$ be a pair of honestly generated keys, and $(m, \mathcal{P}_{\Delta}, \sigma)$ be any tuple message/program/tag with $\mathcal{P}_{\Delta} = (\mathcal{P}, \Delta)$ such that $\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}_{\Delta}, \sigma) = \mathsf{accept}$. Then, for every $\mathsf{vk}_{\mathcal{P}} \leftarrow_{\$} \mathsf{VrfyPrep}_{\mathsf{sk}}(\mathcal{P})$, it holds that:

$$\Pr[\mathsf{EffVrfy}_{\mathsf{sk},\mathsf{vk}_{\mathcal{P}}}(\Delta, m, \sigma) = \mathsf{accept}] = 1. \tag{4.12}$$

**Amortized efficiency** Let $\mathcal{P}_{\Delta} = (\mathcal{P}, \Delta)$ be a multi-labeled program, let $(m_1, \ldots, m_n) \in \mathcal{M}^n$ be any vector of messages, and let $t(n)$ be the time required to compute $\mathcal{P}(m_1, \ldots, m_n)$. If $\mathsf{vk}_{\mathcal{P}} \leftarrow \mathsf{VrfyPrep}_{\mathsf{sk}}(\mathcal{P})$, then the time required for $\mathsf{EffVrfy}_{\mathsf{sk},\mathsf{vk}_{\mathcal{P}}}(\Delta, m, \sigma)$ is $\mathcal{O}(1)$ (independent of $n$).

In this efficiency requirement the cost of computing $\mathsf{vk}_{\mathcal{P}}$ is not considered. That is because the same $\mathsf{vk}_{\mathcal{P}}$ can be re-used in many verification operations with the same labeled program $\mathcal{P}$, but many different data sets $\Delta$. Given this, the cost of computing $\mathsf{vk}_{\mathcal{P}}$ is *amortized* over many verifications of the same function on different data sets.

### 4.2.1 Catalano-Fiore

These constructions only support a restricted set of functions. More precisely, they support polynomials $\{f_n\}$ over $\mathbb{F}$ which have *polynomially bounded degree*, meaning that both the number of variables and the degree of $f_n$ are bounded by some polynomial $p(n)$. The class $\mathcal{VP}$ contains all polynomially bounded degree families of polynomials that are defined by arithmetic circuits of polynomial size and degree.

**Homomorphic MAC from OWFs**

The security of this construction relies only on a PRF (and thus on OWFs). It is very simple and efficient, and allows the homomorphic evaluation of circuits $f\colon \mathbb{Z}_p^n \to \mathbb{Z}_p$ for a prime $p$ of roughly $\lambda$ bits.

This construction is restricted to circuits whose additive gates do not get inputs labeled by constants. Without loss of generality, and when needed, one can use an equivalent circuit where there is a special variable/label for the value of 1, and the MAC of 1 can be published.

The description of the scheme (which we will refer to as CF1) with security parameter $\lambda$ is as follows:

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{ek}, \mathsf{sk})$**.** Let $p$ be a prime of roughly $\lambda$ bits. Choose a seed $K \leftarrow_\$ \{0,1\}^\lambda$ of a PRF function $F_K\colon \{0,1\}^* \to \mathbb{Z}_p$ and a random value $\alpha \leftarrow_\$ \mathbb{Z}_p$. Output $\mathsf{ek} = p$ and $\mathsf{sk} = (K, \alpha)$. The message space $\mathcal{M}$ is defined as $\mathbb{Z}_p$.

$\mathsf{Auth}_{\mathsf{sk}}(\tau, m) \to \sigma$**.** To authenticate a message $m \in \mathbb{Z}_p$ with label $\tau \in \{0,1\}^\lambda$, compute $r_\tau = F_K(\tau)$, set $y_0 = m$, $y_1 = (r_\tau - m)/\alpha \bmod p$ and output $\sigma = (y_0, y_1)$. $y_0, y_1$ are the coefficients of a degree-1 polynomial $y(x)$, where $y(0)$ evaluates to $m$ and $y(\alpha)$ evaluates to $r_\tau$.

In this construction, tags $\sigma$ will be interpreted as polynomials $y \in \mathbb{Z}_p[x]$ of degree $d \geq 1$ in some (unknown) variable $x$, i.e., $y(x) = \sum_i y_i x^i$ (the value of $d$ is increased by successive calls to Eval).

$\mathsf{Eval}_{\mathsf{ek}}(f, \vec{\sigma}) \to \sigma'$**.** Receives an arithmetic circuit $f\colon \mathbb{Z}_p^n \to \mathbb{Z}_p$ and a vector of tags $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$. Basically, Eval consists in evaluating the circuit $f$ on the tags $\vec{\sigma}$ instead of evaluating it on messages. But since the values of $\sigma_i$'s are not valid messages in $\mathbb{Z}_p$, but rather polynomials in $\mathbb{Z}_p[x]$, it is necessary to specify how the evaluation should be done.

Eval proceeds gate-by-gate, and at each gate $g$, given two tags $\sigma_1, \sigma_2$ (or a tag $\sigma_1$ and a constant $c \in \mathbb{Z}_p$), it runs the sub-routine $\sigma' \leftarrow \mathsf{GateEval}_{\mathsf{ek}}(g, \sigma_1, \sigma_2)$ that returns a new tag $\sigma'$, which is then passed as input to the next gate in the circuit. When the last gate of the circuit is reached, Eval outputs the tag vector $\sigma'$ obtained by running GateEval on the last gate. GateEval is described as follows:

$\mathsf{GateEval}_{\mathsf{ek}}(g, \sigma_1, \sigma_2) \to \sigma'$**:** Let $\sigma_i = \vec{y}^{(i)} = (y_0^{(i)}, \ldots, y_{d_i}^{(i)})$ for $i = 1, 2$ and $d_i \geq 1$.

If $g = +$, then:

- let $d = \max(d_1, d_2)$. It is assumed that $d_1 \geq d_2$, so $d = d_1$.

- Compute the coefficients $(y_0, \ldots, y_d)$ of the polynomial $y(x) = y^{(1)}(x) + y^{(2)}(x)$. This can be done efficiently by adding the two vectors of coefficients such that $\vec{y} = \vec{y}^{(1)} + \vec{y}^{(2)}$ ($\vec{y}^{(2)}$ is eventually padded with zeros in positions $d_1 \ldots d_2$).

If $g = \times$, then:

- let $d = d_1 + d_2$.
- Compute the coefficients $(y_0, \ldots, y_d)$ of the polynomial $y(x) = y^{(1)}(x) * y^{(2)}(x)$ using the convolution operator $*$, i.e., $\forall j = 0, \ldots, d : y_j = \sum_{i=0}^{j} y_i^{(1)} \cdot y_{j-i}^{(2)}$.

If $g = \times$ and one of the inputs[5] is a constant $c \in \mathbb{Z}_p$, then:

- let $d = d_1$.
- Compute the coefficients $(y_0, \ldots, y_d)$ of the polynomials $y(x) = c \cdot y^{(1)}(x)$.

Finally, GateEval returns $\sigma' = (y_0, \ldots, y_d)$. The size of a tag grows only after the evaluation of a $\times$ gate (where both inputs are not constants). After the evaluation of a circuit $f$, it holds that $|\sigma'| = \deg(f) + 1$.

$\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}, \sigma) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$. Let $\mathcal{P} = f(\tau_1, \ldots, \tau_n)$ be a labeled program , $m \in \mathbb{Z}_p$ and $\sigma = (y_0, \ldots, y_d)$ be a tag for some $d \geq 1$. Verification is done as follows:

- If $y_0 \neq m$, then output reject. Otherwise, continue.
- For every input wire of $f$ with label $\tau$ compute $r_\tau = F_K(\tau)$. Then compute $\rho = f(r_{\tau_1}, \ldots, r_{\tau_n})$, and use $\alpha$ to check if

$$\rho \stackrel{?}{=} \sum_{i=0}^{d} y_i \alpha^i \tag{4.13}$$

If this is true, then output accept. Otherwise, output reject[6].

**Efficiency** The generation of a tag with Auth is extremely efficient as it only requires one PRF evaluation (e.g., one AES evaluation).

Eval's complexity mainly depends on the cost of evaluating the circuit $f$, and the additional overhead due the GateEval sub-routine and to that the tag's size grows with the degree of the circuit. With a circuit of degree $d$, in the worst case, this overhead is going to be $\mathcal{O}(d)$ for addition gates, and $\mathcal{O}(d \log d)$ for multiplication gates[7]

The cost of Vrfy is basically the cost of computing $\rho$ plus the cost of computing $\sum_{i=0}^{d} y_i \alpha^i$, or $\mathcal{O}(|f| + d)$.

[5] Usually, $\sigma_2$ is the constant.

[6] If using an identity program $\mathcal{I}_\tau$, the Vrfy just checks that $r_\tau = y_0 + y_1 \cdot \alpha$ and $y_0 = m$.

[7] Algorithms based on FFT can be used to efficiently compute the convolution. See Chapter 5 for details.

**Correctness** Essentially, correctness follows from the special property of the tags generated by Auth, i.e., that $y(0) = m$ and $y(\alpha) = r_\tau$. In particular, this property is preserved when evaluating the circuit $f$ over tags $\sigma_1, \ldots, \sigma_n$.

**Security** The security of this scheme is based on the following theorem.

**Theorem 4.11.** *If $F$ is a PRF, then the previously defined homomorphic MAC scheme is secure.*

### Compact Homomorphic MAC

In CF1, the tags' size grows linearly with the degree of the evaluated circuit. This may be acceptable in some cases, e.g., circuits evaluating constant-degree polynomials, but it may become impractical in other situations, e.g., when the degree is greater that the input size of the circuit.

To overcome this problem, a second scheme (CF2) was proposed by Catalano and Fiore that is almost as efficient as the previous one. An interesting property of this second scheme is that the tags are now of constant size. But to achieve this a few restrictions arise. First, there is a fixed a-priori bound $D$ on the degree of allowed circuits. Second, the homomorphic evaluation has to be done in a "single shot" i.e., the authentication tags obtained from Eval cannot be used again to compose with other tags.

Because of these restrictions, another interesting property, *local composition*, is achieved. With this, one can keep a locally non-succinct version of the tag that allows for arbitrary composition. Later, when it comes to send an authentication tag to the verifier, one can securely compress such non-succinct tag in a compact one of constant-size.

Besides a PRF, the security of this scheme relies on a re-writing of the *$\ell$-Diffie-Hellman Inversion* problem. Basically, the definition of $\ell$-Diffie-Hellman Inversion is that one cannot compute $g^{x^{-1}}$ given $g, g^{\alpha}, \ldots, g^{\alpha^D}$. The re-write for this scheme states that one cannot compute $g$ given values $g^{\alpha}, \ldots, g^{\alpha^D}$.

The description of the scheme (which we refer to as CF2) with security parameter $\lambda$ is as follows:

$\mathsf{KeyGen}(1^\lambda, D) \to (\mathsf{ek}, \mathsf{sk})$**:** Let $D = \mathsf{poly}(\lambda)$ be an upper-bound so that the scheme supports circuits of degree at most $D$. Generate a group $\mathbb{G}$ of order $p$ where $p$ is a prime of roughly $\lambda$ bits, and choose a random generator $g \leftarrow_\$ \mathbb{G}$. Choose a seed $K$ of a PRF $F_K \colon \{0,1\}^* \to \mathbb{Z}_p$ and a random value $\alpha \leftarrow_\$ \mathbb{Z}_p$. For $i = 1, \ldots, D$ compute $h_i = g^{\alpha^i}$. Output $\mathsf{ek} = (h_1, \ldots, h_D)$ and $\mathsf{sk} = (K, g, \alpha)$. The message space is defined as $\mathbb{Z}_p$.

$\mathsf{Auth}_{\mathsf{sk}}(\tau, m) \to \sigma$**:** This algorithm is exactly the same as the one from the previous construction. Returns a tag $(y_1, y_2) \in \mathbb{Z}_p^2$.

$\mathsf{Eval}_{\mathsf{ek}}(f, \vec{\sigma}) \to \sigma'$: Takes as input an arithmetic circuit $f\colon \mathbb{Z}_p^n \to \mathbb{Z}_p$ and a vector of tags $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$ such that each $\sigma_i \in \mathbb{Z}_p^2$ (i.e., it is a constant-sized tag for a degree-1 polynomial). First, proceed exactly as in the first scheme to obtain the coefficients $(y_0, \ldots, y_d)$. If $d = 1$ then return $\sigma = (y_0, y_1)$. Otherwise, compute $\Lambda = \prod_{i=1}^{d} h_i^{y_i}$ and return $\sigma = \Lambda$.

$\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}$: Let $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$ be a labeled program, $m \in \mathbb{Z}_p$ and $\sigma$ be a tag of either the form $(y_0, y_1) \in \mathbb{Z}_p^2$ or $\Lambda \in \mathbb{G}$. First, proceed as in the first scheme to compute $\rho$ (i.e., evaluate the circuit with inputs $(r_{\tau_1}, \ldots, r_{\tau_n})$). If $\mathcal{P}$ computes a polynomial of degree 1, then proceed exactly as in the first scheme. Otherwise, use $g$ to check whether the following holds:

$$g^{\rho} \stackrel{?}{=} g^{m} \cdot \Lambda \tag{4.14}$$

If everything is satisfied, then output accept. Otherwise, output reject.

**Efficiency** This scheme is practically as efficient as CF1. Both the tagging and verification algorithms have exactly the same complexity. In the evaluation, the complexity is still dominated by the cost of evaluating the circuit and the cost of the sub-routine GateEval.

**Correctness** Correctness is achieved much like as in CF1. Additionally, the Equation 4.14 is essentially the same as checking that $\rho \stackrel{?}{=} \sum_{i=0}^{d} y_i \alpha^i$, which is the same as Equation 4.13.

**Security** The security of this scheme is based on the following theorem.

**Theorem 4.12.** *If $F$ is a PRF and the $(D-1)$-Diffie Hellman Inversion Assumption holds in $\mathbb{G}$, then the previously defined homomorphic MAC scheme is secure.*

**Local composition** As it was already mentioned, CF2 introduces local composition. This allows us to locally keep the large version of the tag (the polynomial $y$ with its $d+1$ coefficients), but the compact version $\Lambda$ is the one that is sent to the verifier. By keeping this large $y$, it is possible to have composition as in the previous scheme. This is particularly interesting for applications where not too many parties are involved, as it allows for succinct tags and local composition of partial computations at the same time.

### 4.2.2 Backes-Fiore-Reischuk

Both CF and GW schemes suffer from an inefficient verification algorithm. The time to execute Vrfy is proportional to that of executing the program $\mathcal{P}$, i.e., evaluating the circuit $f$. Backes, Fiore, and

Reischuk solved this problem and additionally, they considered the case of performing computations over very large sets of inputs. In this case, besides the already mentioned requirements of security and efficiency, it is desirable to achieve *input-independent efficiency*, meaning that verifying the correctness of a computation $f(m_1, \ldots, m_n)$ requires time *independent* of input size $n$. Additionally, two more crucial requirements should be achieved: *unbounded storage*, meaning that the size of the outsourced data should not be fixed a-priori; and *function independence*, meaning that a client should be able to outsource its data without having to know in advance what functions she will be delegating later.

Input-independent efficiency is achieved in the amortized model: after a first computation with cost $|f|$, the client can verify every subsequent evaluation of $f$ in constant time. By fulfilling unbounded storage and function independence, outsourcing data and function delegation are completely decoupled: a client can continuously outsource data, and the delegated functions do not have to fixed a-priori.

Their construction is also limited to support a restricted set of functions, namely arithmetic circuits of degree up to 2. However, even with this restricted set it is possible to compute a wide range of interesting statistical functions, like counting, summation, (weighted) average, arithmetic mean, standard deviation, variance, co-variance, weighted variance with constant weights, quadratic mean (RMS), mean squared error (MSE), the Pearson product-moment correlation coefficient, the coefficient of determination ($R^2$), and the least squares fit of a data set $\{(x_i, v_i)\}_{i=1}^n$.

In the verification algorithm from both CF constructions it is always necessary to compute $\rho = f(r_{\tau_1}, \ldots, r_{\tau_n})$. One way to accelerate this process would be to, after the first computation of $\rho$, to re-use it in order to verify other computations using $f$. However, this involves the re-use of labels, which is clearly forbidden by the security definition of CF. The security of their scheme completely breaks down in the presence of label re-use.

The proposed solution by Backes, Fiore, and Reischuk to solve this critical problem involves two new ideas. The first one allows the partial, but safe, re-use of labels. This is accomplished with the introduction of multi-labels as previously defined. The other one is a new construction of a PRF that allows the pre-computation of a piece of label-independent information $\omega_f$ that can be re-used to efficiently compute $\rho$.

Before presenting the actual construction, we introduce some definitions and utilities necessary to build a PRF with efficient (amortized) verification. We start by presenting some basic definitions regarding the homomorphic evaluation of arithmetic circuits over values defined in some appropriate set $\mathcal{J} \neq \mathcal{M}$. Usually, $f\colon \mathbb{Z}_p^n \to \mathbb{Z}_p$, but sometimes the message space may be defined in $\mathcal{J}$, and so it is necessary to obtain an equivalent algorithm that evaluates messages from $\mathcal{J}$ over $f$.

**Homomorphic Evaluation over Polynomials** Consider that $\mathcal{J}_{\mathsf{poly}} = \mathbb{Z}_p[x_1, \ldots, x_m]$ is the ring of polynomials in variables $x_1, \ldots, x_m$ over $\mathbb{Z}_p$. For every tuple $\vec{a} = (a_1, \ldots, a_m) \in \mathbb{Z}_p^m$, let $\phi_{\vec{a}} \colon \mathcal{J}_{\mathsf{poly}} \to \mathbb{Z}_p$ be the function defined as $\phi_{\vec{a}}(y) = y(a_1, \ldots, a_m)$ for any $y \in \mathcal{J}_{\mathsf{poly}}$. $\phi_{\vec{a}}$ is a homomorphism from $\mathcal{J}_{\mathsf{poly}}$ to $\mathbb{Z}_p$. Given an arithmetic circuit $f \colon \mathbb{Z}_p^n \to \mathbb{Z}_p$, there exists another structurally equivalent circuit $\hat{f} \colon \mathcal{J}_{\mathsf{poly}}^n \to \mathcal{J}_{\mathsf{poly}}$ such that $\forall y_1, \ldots, y_n \in \mathcal{J}_{\mathsf{poly}} \colon \phi_{\vec{a}}(\hat{f}(y_1, \ldots, y_n)) = f(\phi_{\vec{a}}(y_1), \ldots, \phi_{\vec{a}}(y_n))$. The only difference is that in every gate the operation in $\mathbb{Z}_p$ is replaced by the corresponding operation in $\mathbb{Z}_p[x_1, \ldots, x_m]$.

More precisely, for every positive integer $m \in \mathbb{N}$ and a given $f$, the computation of $\hat{f}$ on $(y_1, \ldots, y_n) \in \mathcal{J}_{\mathsf{poly}}^n$ can be defined as $\mathsf{PolyEval}(m, f, y_1, \ldots, y_n) \to \mathbb{Z}_p$. For every gate $f_g$, on input two polynomials $y_1, y_2 \in \mathcal{J}_{\mathsf{poly}}$, proceeds as follows: if $f_g = +$, it outputs $y = y_1 + y_2$ (adds all coefficients component-wise); if $f_g = \times$, it outputs $y = y_1 \cdot y_2$ (uses the convolution operator on the coefficients). Notice that every $\times$ gate increases the degree $d$ of $y$, as well as its number of coefficients. If $y_1, y_2$ have degree $d_1, d_2$ respectively, the degree of $y = y_1 \cdot y_2$ is $d_1 + d_2$.

**Bilinear Groups** Let $\mathcal{G}(1^\lambda)$ be an algorithm that on input the security parameter $1^\lambda$, outputs the description of a bilinear group $\mathsf{bgpp} = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ where $\mathbb{G}$ and $\mathbb{G}_T$ are groups of the same order $p > 2^\lambda$, $g \in \mathbb{G}$ is a generator and $e \colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is an efficiently computable bilinear map, or pairing function. $\mathcal{G}$ is a *bilinear group generator*.

**Homomorphic Evaluation over Bilinear Groups** Let $\mathsf{bgpp} = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ be the description of a bilinear group as previously defined. With a fixed generator $g \in \mathbb{G}$, then $\mathbb{G} \cong (\mathbb{Z}_p, +)$ (i.e., $\mathbb{G}$ and the additive group $(\mathbb{Z}_p, +)$ are isomorphic) by considering the isomorphism $\phi_g(x) = g^x$ for every $x \in \mathbb{Z}_p$. Similarly, by the property of the pairing function $e$, $\mathbb{G}_T \cong (\mathbb{Z}_p, +)$ by considering the isomorphism $\phi_{g_T}(x) = e(g, g)^x$. Since both $\phi_g$ and $\phi_{g_T}$ are isomorphisms, there also exist the corresponding inverses $\phi_g^{-1} \colon \mathbb{G} \to \mathbb{Z}_p$ and $\phi_{g_T}^{-1} \colon \mathbb{G}_T \to \mathbb{Z}_p$, even though these are not known to be efficiently computable.

For every arithmetic circuit $f \colon \mathbb{Z}_p^n \to \mathbb{Z}_p$ of degree at most 2, the $\mathsf{GroupEval}(f, X_1, \ldots, X_n)$ algorithm homomorphically evaluates $f$ with inputs in $\mathbb{G}$ and outputs in $\mathbb{G}_T$.

Basically, given a circuit $f$ of degree at most 2, and given an $n$-tuple of values $(X_1, \ldots, X_n) \in \mathbb{G}^n$, GroupEval proceeds gate-by-gate as follows: if $f_g = +$, it uses the group operation in $\mathbb{G}$ or in $\mathbb{G}_T$; if $f_g = \times$, it uses the pairing function, thus "lifting" the result to the group $\mathbb{G}_T$. By using only circuits of degree at most 2, the multiplication is well defined.

GroupEval achieves the desired homomorphic property:

**Theorem 4.13.** *Let* $\mathsf{bgpp} = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ *be the description of bilinear groups. Then, the algorithm* $\mathsf{GroupEval}$ *satisfies:* $\forall (X_1, \ldots, X_n) \in \mathbb{G}^n$: $\mathsf{GroupEval}(f, X_1, \ldots, X_n) = e(g,g)^{f(x_1,\ldots,x_n)}$ *for the unique values* $\{x_i\}_{i=1}^n \in \mathbb{Z}_p$ *such that* $X_i = g^{x_i}$.

The proof of Theorem 4.13 can be found in [BFR13].

**PRFs with Amortized Closed-Form Efficiency** A *closed-form efficient* PRF is just like a standard PRF, plus an additional efficiency property. Consider a computation $\mathsf{Comp}(R_1, \ldots, R_n, \vec{z})$ which takes random inputs $R$ and arbitrary inputs $\vec{z}$, and runs in time $t(n, |\vec{z}|)$. By considering the case where each $R_i = \mathsf{F}_K(\mathsf{L}_i)$, then the PRF $\mathsf{F}$ is said to satisfy closed-form efficiency for $(\mathsf{Comp}, \vec{\mathsf{L}})$ if, with the knowledge of the seed $K$, one can compute $\mathsf{Comp}(\mathsf{F}_K(\mathsf{L}_1), \ldots, \mathsf{F}_K(\mathsf{L}_n), \vec{z})$ in time strictly less than $t$. The main idea is that in the pseudo-random case all $R_i$ values have a shorter closed-form representation (as a function of $K$), and this might also allow for the existence of a shorter closed-form representation of the computation $\mathsf{Comp}$.

From the above considerations it possible to define a new property for PRFs: *amortized close-form efficiency*. The idea is to address computations where each $R_i$ is generated as $\mathsf{F}_K(\Delta, \tau_i)$. All the inputs of $\mathsf{F}$ share the same data set label $\Delta$. Then, $\mathsf{F}$ satisfies amortized closed-form efficiency if it is possible to compute $\ell$ computations $\{\mathsf{Comp}(\mathsf{F}_K(\Delta_j, \tau_1), \ldots, \mathsf{F}_K(\Delta_j, \tau_n), \vec{z})\}_{j=1}^{\ell}$ in time strictly less than $\ell \cdot t$.

A PRF is defined as follows:

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pp}, K)$**:** Given a security parameter, output a secret key $K$ and some public parameters $\mathsf{pp}$ that specify domain $\mathcal{X}$ and range $\mathcal{R}$ of the function.

$\mathsf{F}_K(x) \to R$**:** Receives $x \in \mathcal{X}$ and uses the secret key $K$ to compute a value $R \in \mathcal{R}$.

A PRF must satisfy the pseudo-randomness property. More precisely, $(\mathsf{KeyGen}, \mathsf{F})$ is *secure* if for every PPT adversary $\mathcal{A}$ it holds:

$$|\Pr[\mathcal{A}^{\mathsf{F}_K(\cdot)}(1^\lambda, \mathsf{pp}) = 1] - \Pr[\mathcal{A}^{\phi(\cdot)}(1^\lambda, \mathsf{pp}) = 1]| \leq \mathrm{negl}(\lambda) \tag{4.15}$$

$(K, \mathsf{pp}) \leftarrow_{\$} \mathsf{KeyGen}(1^\lambda)$, and $\phi\colon \mathcal{X} \to \mathcal{R}$ is a random function.

**Definition 4.14.** *Consider a computation* $\mathsf{Comp}(R_1, \ldots, R_n, z_1, \ldots, z_m)$ *with random* $n$ *input values and* $m$ *arbitrary input values.* $\mathsf{Comp}$ *takes time* $t(n, m)$. *Let* $\vec{\mathsf{L}} = (\mathsf{L}_1, \ldots, \mathsf{L}_n)$ *be arbitrary values in the domain* $\mathcal{X}$ *of* $\mathsf{F}$ *such that* $\mathsf{L}_i = (\Delta, \tau_i)$. *A PRF (*$\mathsf{KeyGen}$*,* $\mathsf{F}$*) satisfies* amortized closed-form efficiency *for* $(\mathsf{Comp}, \vec{\mathsf{L}})$ *if there exist algorithms* $\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{Comp},\vec{\tau}}$ *and* $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}$ *such that:*

1. *Given* $\omega \leftarrow \mathsf{CFEval}^{\mathsf{off}}_{\mathsf{Comp},\vec{\tau}}(K, \vec{z})$*, then*

$$\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}(K, \omega) = \mathsf{Comp}(\mathsf{F}_K(\Delta, \tau_1), \ldots, \mathsf{F}_K(\Delta, \tau_n), z_1, \ldots, z_m) \tag{4.16}$$

2. *The running time of* $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}(K, \omega)$ *is* $o(t)$.

The $\omega$ obtained from the offline computation $\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{Comp},\tau}$ does not depend on data set label $\Delta$, which means that it can be re-used in further online computations $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}(K, \omega)$ to compute $\mathsf{Comp}(\mathsf{F}_K(\Delta, \tau_1), \ldots, \mathsf{F}_K(\Delta, \tau_n), \vec{z})$ for many different $\Delta$'s. Because of this, the efficiency property puts a restriction only on $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}$ in order to capture the idea of achieving efficiency in an amortized sense when considering many evaluations over of $\mathsf{Comp}(\mathsf{F}_K(\Delta, \tau_1), \ldots, \mathsf{F}_K(\Delta, \tau_n), \ldots, \vec{z})$, with different data set label $\Delta$ in each evaluation. Essentially, one can pre-compute $\omega$ once, and then use it to run the online phase as many times as needed, almost for free.

The structure of Comp may impose some restrictions on the range $\mathcal{R}$ of the PRF, and due to the pseudo-randomness property, the output distribution of $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{Comp},\Delta}(K, \mathsf{CFEval}^{\mathsf{off}}_{\mathsf{Comp},\vec{\tau}}(K, \vec{z}))$ is computationally indistinguishable from the output distribution of $\mathsf{Comp}(R_1, \ldots, R_n, \vec{z})$.

**Amortized Closed-Form Efficient PRF** Before introducing the description of BFR scheme, it is necessary to define this new and more efficient PRF, namely a PRF with amortized closed-form efficiency for GroupEval. This PRF uses two generic PRFs which map binary strings to integers in $\mathbb{Z}_p$, and a weak PRF whose security relies on the Decision Linear assumption introduced by Boneh, Boyen, and Shacham [BBS04], and is presented below:

**Definition 4.15.** *Let* $\mathcal{G}$ *bilinear group generator, and* $\mathsf{bgpp} = (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow_\$ \mathcal{G}(1^\lambda)$*. Let* $g_0, g_1, g_2 \leftarrow_\$ \mathbb{G}$ *and* $r_0, r_1, r_2 \leftarrow_\$ \mathbb{Z}_p$ *be chosen uniformly at random. The advantage of a PPT adversary* $\mathcal{A}$ *in solving the* Decision Linear *problem is*

$$\begin{aligned}\mathbf{Adv}^{DLin}_{\mathcal{A}}(\lambda) = |\Pr[\mathcal{A}(\mathsf{bgpp}, g_0, g_1, g_2, g_1^{r_1}, g_2^{r_2}, g_0^{r_1+r2}) = 1] - \\ \Pr[\mathcal{A}(\mathsf{bgpp}, g_0, g_1, g_2, g_1^{r_1}, g_2^{r_2}, g_0^{r_0}] = 1|\end{aligned} \tag{4.17}$$

*The Decision Linear assumption holds for* $\mathcal{G}$ *if for every PPT adversary* $\mathcal{A}$ *if* $\mathbf{Adv}^{DLin}_{\mathcal{A}}(\lambda) \leq \mathrm{negl}(\lambda)$.

**Syntax** The syntax of an amortized closed-form efficient PRF is as follows:

$\mathsf{KeyGen}(1^\lambda) \rightarrow (\mathsf{pp}, K)$**:** Let $\mathsf{bgpp} = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ be the description of bilinear groups $\mathbb{G}$ and $\mathbb{G}_T$ having the same prime order $p > 2^\lambda$ and such that $g \in \mathbb{G}$ is a generator, and $e\colon \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$

is an efficiently computable bilinear map. Choose two seeds $K_1, K_2$ for a family of PRFs $\mathsf{F}'_{K_i}$: $\{0,1\}^* \to \mathbb{Z}_p^2$ for $i = 1, 2$. Output $K = (\mathsf{bgpp}, K_1, K_2)$ and $\mathsf{pp} = \mathsf{bgpp}$.

$\mathsf{F}_K(x) \to R$**:** Let $x = (\Delta, \tau) \in \mathcal{X}$. To compute $R \in \mathbb{G}$, generate $(u, v) \leftarrow \mathsf{F}'_{K_1}(\tau)$ and $(a, b) = \mathsf{F}'_{K_2}(\Delta)$, and output $R = g^{ua+vb}$.

The previous PRF is pseudo-random. The proof of the following theorem can be found in [BFR13].

**Theorem 4.16.** *If* $\mathsf{F}'$ *is a PRF and the Decision Linear assumptions holds for* $\mathcal{G}$*, then the function* $(\mathsf{KeyGen}, \mathsf{F})$ *presented above is a PRF.*

**Amortized Closed-Form Efficiency for** GroupEval The previously defined PRF satisfies amortized closed-form efficiency for $(\mathsf{GroupEval}, \vec{\mathsf{L}})$. Recall that $\mathsf{GroupEval}\colon f \times \mathbb{G}^n \to \mathbb{G}_T$, with arithmetic circuit $f\colon \mathbb{Z}_p^n \to \mathbb{Z}_p$ and $R_1, \ldots, R_n \in \mathbb{G}$ random values. $\vec{\mathsf{L}}$ is a vector $(\mathsf{L}_1, \ldots, \mathsf{L}_n)$ with each $\mathsf{L}_i = (\Delta, \tau_i) \in \mathcal{X}$.

$\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{GroupEval},\vec{\tau}}(K, f) \to \omega_f$**:** Let $K = (\mathsf{bgpp}, K_1, K_2)$ be a secret key generated by $\mathsf{KeyGen}(1^\lambda)$ of a closed-form efficient PRF. Compute $\{(u_i, v_i) \leftarrow \mathsf{F}'_{K_1}(\tau_i)\}_{i=1}^n$ and set $\vec{\rho_i} = (0, u_i, v_i)$. Basically, $\vec{\rho_i}$ are the coefficients of a degree-1 polynomial $\rho_i(x_1, x_2)$ in two unknown variables $x_1, x_2$. Then, run $\mathsf{PolyEval}(2, f, \rho_1, \ldots, \rho_n)$ to compute the coefficients $\vec{\rho}$ of a polynomial $\rho(x_1, x_2)$ such that $\forall x_1, x_2 \in \mathbb{Z}_p\colon \rho(x_1, x_2) = f(\rho_1(x_1, x_2), \ldots, \rho_n(x_1, x_2))$. Output $\omega_f = \vec{\rho}$.

$\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{GroupEval},\Delta}(K, \omega_f) \to W$**:** Let $K = (\mathsf{bgpp}, K_1, K_2)$ be a secret key and $\omega_f = \vec{\rho}$ the result of the previous algorithm. Generate $(a, b) \leftarrow \mathsf{F}'_{K_2}(\Delta)$, and then use the coefficients $\vec{\rho}$ to compute $w = \rho(a, b)$. Output $W = e(g, g)^w$.

**Theorem 4.17.** *Let* $\vec{\mathsf{L}} = (\mathsf{L}_1, \ldots, \mathsf{L}_n)$ *be such that* $\mathsf{L}_i = (\Delta, \tau_i) \in \mathcal{X}$ *and let* $t$ *be the running time of* GroupEval*. Then the PRF* $(\mathsf{KeyGen}, \mathsf{F})$*, extended with the algorithms* $\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{GroupEval},\vec{\tau}}$ *and* $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{GroupEval},\Delta}$ *satisfies* amortized closed-form efficiency for $(\mathsf{GroupEval}, \vec{\mathsf{L}})$ *according to Definition 4.14, with running times of* $\mathcal{O}(t)$ *and* $\mathcal{O}(1)$ *for* $\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{GroupEval},\vec{\tau}}$ *and* $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{GroupEval},\Delta}$*, respectively.*

The proof can be found in [BFR13].

**Homomorphic MAC with Efficient Verification** Having defined the PRF that allows to more efficiently compute the $\rho$ from the verification algorithm, we can now present the construction of an homomorphic MAC scheme with an efficient verification algorithm.

BFR scheme works for circuits whose additive gates do not get inputs labeled by constants. And just like in the CF, without loss of generality, one can use an equivalent circuit where there is a special

variable/label for the value of 1, and the MAC of 1 can be published. The description of the EVH-MAC construction is as follows:

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{ek}, \mathsf{sk})$**:** Run $\mathsf{bgpp} \leftarrow_\$ \mathcal{G}(1^\lambda)$ to generate the description of bilinear groups with $\mathsf{bgpp} = (p, \mathbb{G}, \mathbb{G}_T, e, g)$. The message space $\mathcal{M}$ is $\mathbb{Z}_p$. Choose a random value $\alpha \leftarrow_\$ \mathbb{Z}_p$, and run $(K, \mathsf{pp}) \leftarrow_\$ \mathsf{KeyGen}_(1^\lambda)$ to obtain seed $K$ of a PRF function $\mathsf{F}_K \colon \{0,1\}^* \times \{0,1\}^* \to \mathbb{G}$. Output $\mathsf{sk} = (\mathsf{bgpp}, \mathsf{pp}, K, \alpha)$ and $\mathsf{ek} = (\mathsf{bgpp}, \mathsf{pp})$.

$\mathsf{Auth}_{\mathsf{sk}}(\mathsf{L}, m) \to \sigma$**:** Receives a multi-label $(\Delta, \tau) \in (\{0,1\}^\lambda)^2$ and a message $m \in \mathcal{M}$. Compute $R \leftarrow \mathsf{F}_K(\Delta, \tau)$ and set $y_0 = m$ and $Y_1 = (R \cdot g^{-m})^{1/\alpha}$. Output the tag $(y_0, Y_1) \in \mathbb{Z}_p \times \mathbb{G}$.

If we consider the $y_1 \in \mathbb{Z}_p$ to be the unique value such that $Y_1 = g^{y_1}$, then $(y_0, y_1)$ are simply the coefficients of a degree-1 polynomial $y(x)$ such that $y(0) = m$ and $y(\alpha) = \phi_g^{-1}(R)$.

$\mathsf{Eval}_{\mathsf{ek}}(f, \vec{\sigma} \to \sigma')$**:** Receives a vector of tags $\sigma_1, \ldots, \sigma_n$ and an arithmetic circuit $f \colon \mathbb{Z}_p^n \to \mathbb{Z}_p$. Just as in CF scheme, the values of $\sigma_i$'s are not valid messages in $\mathbb{Z}_p$, and so it is necessary to specify the sub-routine GateEval. At a given gate $f_g$, given two tags $\sigma_1, \sigma_2$, GateEval proceeds as follows:

$\mathsf{GateEval}_{\mathsf{ek}}(f_g, \sigma_1, \sigma_2) \to \sigma'$**:** Let $\sigma_i = (y_0^{(i)}, Y_1^{(i)}, \hat{Y}_2^{(i)}) \in \mathbb{Z}_p \times \mathbb{G} \times \mathbb{G}_T$ for $i = 1, 2$. Assume that $\hat{Y}_2^{(i)} = 1$ whenever it is not defined.

If $f_g = +$, then compute $(y_0, Y_1, \hat{Y}_2)$ as:

- $y_0 = y_0^{(1)} + y_0^{(2)}$
- $Y_1 = Y_1^{(1)} \cdot Y_1^{(2)}$
- $\hat{Y}_2 = \hat{Y}_2^{(1)} \cdot \hat{Y}_2^{(2)}$

If $f_g = \times$, then compute $(y_0, Y_1, \hat{Y}_2)$ as:

- $y_0 = y_0^{(1)} \cdot y_0^{(2)}$
- $Y_1 = (Y_1^{(1)})^{y_0^{(2)}} \cdot (Y_1^{(2)})^{y_0^{(1)}}$
- $\hat{Y}_2 = e(Y_1^{(1)}, Y_1^{(2)})$

  Since this construction assumes that $deg(f) \leq 2$, it can be assumed that $\sigma_i = (y_0^{(i)}, Y_1^{(i)}) \in \mathbb{Z}_p \times \mathbb{G}$ for $i = 1, 2$.

If $f_g = \times$ and one of the two inputs is a constant $c \in \mathbb{Z}_p$ (assume that $\sigma_2$ is the constant), then compute $(y_0, Y_1, \hat{Y}_2)$ as:

- $y_0 = c \cdot y_0^{(1)}$
- $Y_1 = (Y_1^{(1)})^c$
- $\hat{Y}_2 = (\hat{Y}_2^{(1)})^c$

Return $\sigma' = (y_0, Y_1, \hat{Y}_2)$.

$\mathsf{Vrfy}_{\mathsf{sk}}(m, \mathcal{P}_\Delta, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}$**:** Receives a message $m \in \mathcal{M}$, a multi-labeled program $\mathcal{P}_\Delta = (f, \tau_1, \ldots, \tau_n), \Delta)$ and a tag $\sigma = (y_0, Y_1, \hat{Y}_2)$. For $i = 1$ to $n$, compute $R_i \leftarrow \mathsf{F}_K(\Delta, \tau_i)$. Then run $W \leftarrow \mathsf{GroupEval}(f, R_1, \ldots, R_n) \in \mathbb{G}_T$, and check that both:

$$m \stackrel{?}{=} y_0$$
$$W \stackrel{?}{=} e(g,g)^{y_0} \cdot e(Y_1, g)^{\alpha} \cdot (\hat{Y}_2)^{\alpha^2}$$

If both checks are satisfied, output accept. Output reject otherwise.

The previous verification algorithm is still not efficient. The description of EVH-MAC needs to be complemented with VrfyPrep and EffVrfy to achieve efficient verification.

$\mathsf{VrfyPrep}_{\mathsf{sk}}(\mathcal{P}) \to \mathsf{vk}_{\mathcal{P}}$**:** Let $\mathsf{sk} = (\mathsf{bgpp}, \mathsf{pp}, K, \alpha)$. Receives a labeled program $\mathcal{P} = (f, \tau_1, \ldots, \tau_n)$. A concise verification key $\mathsf{vk}_{\mathcal{P}} = \omega$ is computed, with $\omega \leftarrow \mathsf{CFEval}^{\mathsf{off}}_{\mathsf{GroupEval},\vec{r}}(K, f)$.

$\mathsf{EffVrfy}_{\mathsf{sk},\mathsf{vk}_{\mathcal{P}}}(\Delta, m, \sigma) \to \{\mathsf{accept}, \mathsf{reject}\}$**:** Let $sk = (\mathsf{bgpp}, \mathsf{pp}, K, \alpha)$ and $\mathsf{vk}_{\mathcal{P}} = \omega$ and $\sigma = (y_0, Y_1, \hat{Y}_2)$. First, run the online closed-form efficient algorithm of $\mathsf{F}$ for GroupEval to compute $W \leftarrow \mathsf{GroupEval}^{\mathsf{on}}_{\mathsf{GroupEval},\Delta}(K, \omega)$. Then, it runs the same checks as in Vrfy, and returns accept if both checks are satisfied, false otherwise.

**Correctness** EVH-MAC satisfies both authentication and evaluating correctness. The respective proofs can be found in [BFR13].

**Efficient Verification** EVH-MAC satisfies efficient verification.

**Theorem 4.18.** *If* $\mathsf{F}$ *has amortized closed-form efficiency for* $(\mathsf{GroupEval}, \vec{\mathsf{L}})$*, then* EVH-MAC *satisfies efficient verification.*

The verification preparation VrfyPrep runs in time equal to $\mathsf{CFEval}^{\mathsf{off}}_{\mathsf{GroupEval},\vec{r}}$, and the efficient verification EffVrfy runs in time equal to $\mathsf{CFEval}^{\mathsf{on}}_{\mathsf{GroupEval},\Delta}$. From Theorem 4.17, VrfyPrep and EffVrfy run in time $\mathcal{O}(|f|)$ and $\mathcal{O}(1)$, respectively. The full proof can be found in [BFR13].

**Security** The security of EVH-MAC is based on the following theorem.

**Theorem 4.19.** *Let* $\lambda$ *be the security parameter,* $\mathsf{F}$ *be a PRF with security* $\epsilon_{\mathsf{F}}$*, and* $\mathcal{G}$ *be a bilinear group generator. Then, any PPT adversary* $\mathcal{A}$ *making* $Q$ *verification queries has at most probability*

$$\Pr[\mathsf{HomUF} - \mathsf{CMA}_{\mathcal{A},\mathsf{HMAC\text{-}ML}} = 1] \leq 2 \cdot \epsilon_{\mathsf{F}} + \frac{8Q}{p - 2(Q-1)} \tag{4.18}$$

*of breaking the security of* EVH-MAC.

## 4.3 Limitations and open problems

Homomorphic signatures are still far from being a truly practical primitive. The data set size has to be fixed a-priori, and besides the BF scheme with support for polynomials of bounded degree, only linear realizations have been proposed. However, for quadratic and higher degree polynomials, the derived signatures leak information about the original data set. This privacy property is only achieved for linear functions, and therefore, it remains an open problem to extend it for quadratic and higher polynomials.

The security of BF scheme could be strengthened by removing the random oracle model which is used to simulate signatures on a chosen message attacker, but it is not known if that's even possible without losing the homomorphic properties.

In the private key setting, the existence of a fully homomorphic MAC scheme is already known as it was shown by Gennaro and Wichs [GW12]. However, its security model does not allow an adversary to ask for verification queries, and it would be interesting to improve the GW scheme to support an unbounded number of verification queries. Just as with the CF constructions, the complexity of the verification algorithm of a fully homomorphic MAC is not independent of the complexity of a program $\mathcal{P}$. This, and the fact that it relies on FHE "machinery", makes the GW scheme not practical.

The CF and BFR schemes only support a limited class of functions. Even though they allow the computation of many interesting functions, and because of their superior efficiency compared to fully homomorphic MACs, it would be very interesting to extend them to a broader class of functions, i.e., to construct a fully homomorphic MAC for circuits of arbitrary size.

Another interesting open problem is the existence of a *Fully Homomorphic Signature* scheme which could authenticate the computation of any function on signed data. This scheme would be particularly useful to be used alongside a FHE scheme like the one from Gentry [Gen09]. It is also not clear whether processes from Gentry's construction could be applied to signature schemes in order to improve their security and efficiency. While that is not possible, it would be useful to enlarge the set of admissible functions of homomorphic signatures schemes.

# Chapter 5

# Efficient algebraic computation algorithms

Almost all existing cryptographic primitives offering more than the most basic of functionalities heavily relies on algebraic operations. That is why when implementing such primitives one must pay special attention to the way these algebraic operations are performed. The "classical" textbook way may be efficient enough for small (and not so practical) parameters, but when dealing with thousand bit numbers things become noticeably slow.

In CF's homomorphic evaluation, more precisely GateEval is the critical and the slowest operation and thus needs to be implemented as efficiently as possible. This sub-routine is simply an arithmetic circuit evaluation, but instead of using input values from field $\mathbb{F}_p$ , it uses polynomial coefficients from $\mathbb{F}_p[X]$ as a result of previous Auth's or Eval's of $n$ messages. Sum gates and product gates compute the addition and multiplication of two polynomials, respectively.

To the sum gate there isn't much that we can do since the straightforward algorithm takes $\Theta(n)$. On the other hand, the product gate is where a big optimization can be introduced. The straightforward method of multiplication takes $\Theta(n^2)$, and for big enough values of $n$, this becomes extremely slow, and therefore compromises the efficiency of the entire homomorphic scheme. So it is obvious that this is where we have to find some more efficient algorithms in order to reduce this time complexity. As for the multiply by a constant, the straightforward way is $\Theta(n)$ as well. Following on the suggestion by Catalano and Fiore, the product problem can be mitigated by using algorithms based on the fast Fourier Transform (FFT), which can reduce the multiplication of two polynomials to $\Theta(n \log n)$.

Fourier Transforms are widely used in signal processing, and without them we wouldn't have much of the technological developments of today. A signal is given in the *time domain*: as a function mapping time to amplitude. The Fourier analysis allows us to express the signal as a weighted sum of

phase-shifted sinusoids of varying frequencies. The weights and phases associated with the frequencies characterize the signal in the *frequency domain*. Fast Fourier analysis is widely employed in many of our applications of today: audio recording, MP3 audio compression or JPEG image encoding and compression.

Signal processing is just one of the many applications of Fourier Transforms. One other field where it can be used, and the one of interest to us, is where it can be used to perform an efficient multiplication of two polynomials.

## 5.1 Polynomials

A *polynomial* in the variable $x$ over the field $\mathbb{F}$ can be represented as a summation like:

$$P(x) = \sum_{i=0}^{n-1} p_i x^i$$

The $p_i$ values are the *coefficients* of the polynomial, and are drawn from the underlying field $\mathbb{F}$. The degree of the polynomial $P(x)$ is $d$ if its highest nonzero coefficient is $p_d$; we denote it by $\text{degree}(P) = d$. Any integer strictly greater than the degree of the polynomial is the *degree-bound* of that polynomial. The degree of a polynomial of degree-bound $n$ may be any integer between 0 and $n-1$, inclusive. The usual operations over polynomials, and the ones that are of interest to us, are addition and multiplication.

For polynomial addition, if $A(x)$ and $B(x)$ are polynomials of degree-bound $n$:

$$A(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{and} \quad B(x) = \sum_{i=0}^{n-1} b_i x^i,$$

their sum is a polynomial $C(x) = A(x) + B(x)$, also of degree-bound $n$, as:

$$C(x) = \sum_{i=0}^{n-1} c_i x^i, \quad \text{such that } c_i = a_i + b_i \tag{5.1}$$

As a result, it holds that $\text{degree}(C) = \max(\text{degree}(A), \text{degree}(B))$.

The straightforward way of multiplying two polynomials $A(x)$ and $B(x)$ of degree-bound $n$, resulting in the product $C(x) = A(x)B(x)$ of degree bound $2n-1$, is expressed as:

$$C(x) = \sum_{i=0}^{2n-2} c_i x^i, \quad \text{such that each } c_i = \sum_{j=0}^{i} a_j b_{i-j} \tag{5.2}$$

If $A(x)$ and $B(x)$ are of degree-bound $n_a$ and $n_b$, respectively, and since that clearly degree$(C)$ = degree$(A)$ + degree$(B)$, then $C(x)$ is of degree-bound $n_a + n_b - 1$. Because every polynomial of degree-bound $n$ is also of degree-bound $n + 1$, we can also say that $C(x)$ is of degree-bound $n_a + n_b$.

### 5.1.1 Representation

The way we represent polynomials can be directly related to the time it takes to execute some operation over them, especially in the case of multiplication. We now present the two common representations, which are the ones used to obtain a faster polynomial multiplication with FFT.

**Coefficient representation** The most natural and simple representation is the *coefficient representation*, where a polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$ of degree-bound $n$ is represented as a vector of coefficients $a = (a_0, \ldots, a_{n-1})$.

This representation is very convenient for some operations. For instance, the evaluation of the polynomial $A(x)$ at a given point $x_0$ can be done in $\Theta(n)$ time using *Horner's rule*:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-2} + x_0(a_{n-1}))\ldots,))$$

Adding two polynomials in coefficient representation is also very simple: having two vectors $a$ and $b$ representing the polynomials $A(x)$ and $B(x)$, respectively, the addition is simply their vector addition $c = (a_0 + b_0, \ldots, a_{n-1} + b_{n-1})$ which takes $\Theta(n)$ time.

But if we look at polynomial multiplication (with coefficient representation) of two polynomials $A(x)$ and $B(x)$ of degree-bound $n$, we get a multiplication with $\Theta(n^2)$ time. That is because we have to multiply each coefficient of vector $a$ by each coefficient in vector $b$. The resulting vector $c$ is also called the *convolution* of the input vectors $a$ and $b$, denoted as $c = a \otimes b$.

**Point-value representation** A *point-value representation* of a polynomial $A(x)$ of degree-bound $n$ is a set of $n$ *point-value pairs*

$$\{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}$$

such that all $x_i$ are *distinct* and for each $i = 0, \ldots, n-1$ it holds that $y_i = A(x_i)$. A polynomial has many different point-value representations since we can choose any set of $n$ distinct points $x_0, \ldots, x_{n-1}$.

The conversion from coefficient to point-value representation is simple: select $n$ distinct points $x_0, \ldots, x_{n-1}$ and then evaluate every $A(x_i)$ for $i = 0, \ldots, n-1$. With Horner's rule, this evaluation at

$n$ points takes $\Theta(n^2)$ time. But there is another way of performing this conversion in time $\Theta(n \log n)$ if we carefully choose the $n$ points in a certain way, as we will see later in this chapter.

The inverse of polynomial evaluation is *interpolation* – determine the coefficient form of a polynomial given a set of point-value pairs. The following theorem shows that interpolation is well defined when the desired interpolating polynomial must have a degree-bound equal to the given number of point-value pairs.

**Theorem 5.1** ([CLRS09, Theorem 30.1]). *For any set $\{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}$ of $n$ point-value pairs such that all $x_i$ values are distinct, there is a unique polynomial $A(x)$ of degree-bound $n$ such that $y_i = A(x_i)$ for $i = 0, \ldots, n-1$.*

Theorem 5.1 shows the *uniqueness of an interpolating polynomial* (see [CLRS09] for the full proof) and it also describes an algorithm based on solving a set of linear equations. But the fast algorithm to solve those equations has time $\Theta(n^3)$.

A faster algorithm for $n$-point interpolation is based on *Lagrange's formula*:

$$A(x) = \sum_{i=0}^{n-1} y_i \ell_i(x) \quad \text{where } \ell_i(x) = \prod_{\substack{0 \le j \le n-1 \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \tag{5.3}$$

Just like $n$-point evaluation, interpolation with Lagrange's formula takes time $\Theta(n^2)$.

A nice property is that $n$-point evaluation and interpolation are well-defined inverse operations that transform between the two polynomials representations: coefficient and point-value.

Just like in coefficient representation, we can perform many operations on the point-value representation of the polynomial, namely addition and multiplication.

For addition, it follows that if $C(x) = A(x) + B(x)$, then $C(x_i) = A(x_i) + B(x_i)$ for any point $x_i$. More precisely, if we have:

$$A(x) = \{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\} \quad \text{and} \quad B(x) = \{(x_0, y'_0), \ldots, (x_{n-1}, y'_{n-1})\}$$

then a point-value representation for $C(x)$ is:

$$C(x) = \{(x_0, y_0 + y'_0), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

and so the time to addition of two polynomials of degree-bound $n$ in point-value representation is $\Theta(n)$.

Now, and unlike in coefficient representation, polynomial multiplication can be faster. If $C(x) = A(x)B(x)$, then $C(x_i) = A(x_i)B(x_i)$ for any point $x_i$, thus it is now possible to simply point-wise multiply a point-value representation of $A$ by a point-value representation of $B$ to obtain a point-value representation of $C$. But we a have a small problem because it must hold that degree$(C)$ = degree$(A)$+ degree$(B)$ ($C$ is polynomial of degree-bound $2n$), but with a standard point-wise multiplication of point-value representations of $A$ and $B$ yields only $n$ point-value pairs, and we are expected to obtain $2n$ pairs in order to interpolate a polynomial of degree-bound $2n$. So in order to fix this problem, we first must add $n$ extra points to the point-value representations of $A$ and $B$, so that they end up with $2n$ point-value pairs each. Basically, $A$ and $B$ are represented as:

$$A(x) = \{(x_0, y_0), \ldots, (x_{2n-1}, y_{2n-1})\} \quad \text{and} \quad B(x) = \{(x_0, y'_0), \ldots, (x_{2n-1}, y'_{2n-1})\}$$

and the resulting $C(x) = A(x)B(x)$ as:

$$C(x) = \{(x_0, y_0 y'_0), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}$$

Such a multiplication technique takes $\Theta(n)$ time, much less than with the usual coefficient representation.

As for the evaluation of a polynomial in point-value representation at a new point (i.e., computing the value of $A(x_i)$ on a given point $x_i$) there is no known direct technique. The best alternative is to first convert from point-value to coefficient representation, and then evaluating it at the new point.

| | $n$-point Evaluation | Interpolation | Addition | Multiplication |
|---|---|---|---|---|
| Coefficient | $\Theta(n^2)$ | – | $\Theta(n)$ | $\Theta(n^2)$ |
| Point-value | – | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n)$ |

**Table 5.1:** The Evaluation column refers to the conversion of a polynomial in coefficient form to a polynomial in point-value form, while the Interpolation column represents the inverse operation. The Addition times are the same in both representations, but Multiplication is clearly faster with a point-value representation. However, the conversion between both representations is not efficient.

We saw the two common representations of polynomials and some operations that can be executed over them. On Table 5.1 you can find an overview of the execution time of such operations. And as we can see, the time to convert between coefficient form and point-value is very slow. But is there a way to improve these conversions, in order to take advantage of the linear time multiplication method for polynomials in point-value form?

As it turns, yes, it is possible. The trick is to choose the evaluation points (when converting from coefficient to point-value) carefully, and then it is possible to convert between both representations in

only $\Theta(n \log n)$. This result is possible by choosing complex roots of unity as the evaluation points, and then by using Fourier Transform to convert between both representations.

The following is a brief description of the $\Theta(n \log n)$ multiplication for two polynomials $A(x)$ and $B(x)$ of degree-bound $n$, where both input and output are in coefficient form. It is assumed that $n$ is a power of 2; this can always be met if we add high-order zero coefficients.

1. *Double degree bound:* Add $n$ high order zero coefficients to the coefficient representations of $A(x)$ and $B(x)$, so that they become polynomials of degree-bound $2n$.

2. *Evaluate:* Convert the two coefficient representations to point-value representations, using a (Discrete) Fourier Transform with $\Theta(n \log n)$ time.

3. *Point-wise multiply:* Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying them point-wise. The result $C(x)$ contains $2n$ point-value pairs.

4. *Interpolate:* Convert the point-value representation of $C(x)$ to coefficient representation by applying the (inverse Discrete) Fourier Transform with $\Theta(n \log n)$ time. In the end, we have $C(x)$ in coefficient representation.

In the following section we will introduce the Fourier Transforms of (2) and (4).

## 5.2 Fourier Transforms

As stated before, the trick to have fast conversions between coefficient and point-value representations is to carefully choose the evaluation points as complex roots of unit. In this section we show what are they, and we also define the DFT, and then show how FFT computes the DFT and its inverse in time $\Theta(n \log n)$.

From this section on, because we start to use complex numbers, the symbol $j$ is exclusively used to denote $\sqrt{-1}$.

### 5.2.1 Complex roots of unity

A *complex $n$th root of unity* is a complex number $\omega$ such that $\omega^n = 1$. There are exactly $n$ complex $n$th roots of unity:

$$e^{2\pi j \frac{k}{n}} \qquad 0 \leq k < n \tag{5.4}$$

To interpret Equation 5.4, we use Euler's formula that reads as $e^{ju} = \cos u + j \sin u$.

All complex $n$th roots of unity can be written as powers of the *principal $n$th root of unity*, denoted as $\omega_n = e^{2\pi \frac{j}{n}}$. The 7th complex roots of unity can be seen in Figure 5.1.

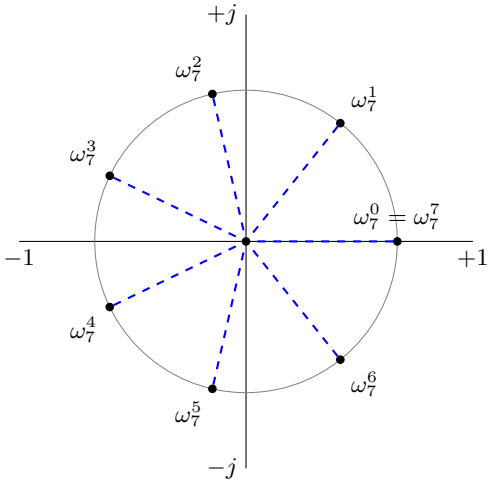



**Figure 5.1:** The values of the 7th complex roots $\omega_7^0, \ldots, \omega_7^6$ in the complex plane, where $\omega_7^7 = e^{2\pi \frac{j}{7}}$ is the principal 7th root of unity.

The $n$ complex $n$th roots of unity $\omega_n^0, \ldots, \omega_n^{n-1}$ form a group under multiplication with the same structure as the additive group $(\mathbb{Z}_n, +)$ modulo $n$, since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^k \omega_n^i = \omega_n^{k+i} = \omega_n^{(k+i) \bmod n}$. Similarly, $\omega_n^{-1} = \omega_n^{n-1}$.

We now present some lemmas to better understand some essential properties of complex $n$th roots of unity. We refer to [CLRS09] for the full proofs.

**Lemma 5.2** (Cancelation lemma [CLRS09, Lemma 30.3])**.** *For any integers $n \geq 0$, $k \geq 0$, and $d > 0$,*

$$\omega_{dn}^{dk} = \omega_n^k \tag{5.5}$$

**Corollary 5.3** ([CLRS09, Corollary 30.4])**.** *For any integer $n > 0$,*

$$\omega_n^{n/2} = \omega_2 = -1 \tag{5.6}$$

**Lemma 5.4** (Halving lemma [CLRS09, Lemma 30.5])**.** *If $n > 0$ is even, then the squares of the $n$ complex $n$th roots of unity are the $n/2$ complex $(n/2)$th roots of unity.*

**Lemma 5.5** (Summation lemma [CLRS09, Lemma 30.6])**.** *For any integer $n \geq 1$ and non-zero integer $k$ not divisible by $n$,*

$$\sum_{i=0}^{n-1} \left(\omega_n^k\right)^i = 0 \tag{5.7}$$

### 5.2.2 Discrete Fourier Transform

We want to evaluate a polynomial $A(x)$ of degree-bound $n$ at the $n$ complex $n$th roots of unity[1]. Assuming that $A(x)$ is given in coefficient form $a = (a_0, \ldots, a_{n-1})$, we want to obtain a vector $y = (y_0, \ldots, y_{n-1})$, where each $y_k$ is of the form:

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{i=0}^{n-1} a_i \omega_n^{ki} \end{aligned} \tag{5.8}$$

The vector $y$ is the *Discrete Fourier Transform* of the coefficient vector $a$, or more succinctly, $y = \mathrm{DFT}_n(a)$. Note that this still takes $\Theta(n^2)$ time.

### 5.2.3 Fast Fourier Transform

By using a *fast Fourier Transform*, which takes advantage of the special properties of the complex roots of unity, it is possible to compute $\mathrm{DFT}_n(a)$ in time $\Theta(n \log n)$. It is assumed that $n$ is a power of 2.

The FFT technique uses a divide-and-conquer strategy by dividing $A(x)$ into two polynomials of degree-bound $n/2$:

$$A^{(0)}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$$
$$A^{(1)}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

Because $A^{(0)}$ contains all the even-indexed coefficients and $A^{(1)}$ contains all the odd-indexed coefficients, we have that:

$$A(x) = A^{(0)}(x^2) + xA^{(1)}(x^2). \tag{5.9}$$

So basically, the problem of evaluating $A(x)$ at the $n$ complex $n$th roots of unity reduces to:

1. evaluate the degree-bound $n/2$ polynomials $A^{(0)}(x)$ and $A^{(1)}(x)$ at the points $(\omega_n^0)^2, \ldots, (\omega_n^{n-1})^2$, and then
2. combine the results according to Equation 5.9.

Because of Lemma 5.4 (halving lemma), the list of points in (1) has not $n$ distinct values but only $n/2$ complex $(n/2)$th roots of unity, with each root occurring exactly twice. This means that we can

[1] In the context of polynomial multiplication, the length $n$ is actually $2n$, but for clarity we use $n$ instead of $2n$ in this section.

divide a $n$-element $\text{DFT}_n$ computation into two $n/2$-element $\text{DFT}_{n/2}$ computations. This is the basis for the definition of the recursive FFT algorithm, which computes the DFT of an $n$-element vector $a = (a_0, \ldots, a_{n-1})$, with $n$ a power of 2. Such algorithm is presented in Algorithm 3.

**Algorithm 3** Recursive FFT.

```
1:  function RECURSIVE-FFT(a)
2:      n ← a.length
3:      if n == 1 then
4:          return a
5:      end if
6:      ω_n ← e^{2πj/n}
7:      ω ← 1
8:      a^(0) ← (a_0, a_2, ..., a_{n-2})
9:      a^(1) ← (a_1, a_3, ..., a_{n-1})
10:     y^(0) ←RECURSIVE-FFT(a^(0))
11:     y^(1) ←RECURSIVE-FFT(a^(1))
12:     for k = 0 to n/2 − 1 do
13:         y_k ← y_k^(0) + ω y_k^(1)
14:         y_{k+(n/2)} ← y_k^(0) − ω y_k^(1)
15:         ω ← ω ω_n
16:     end for
17:     return y
18: end function
```

To analyze the running time of Algorithm 3, note that each invocation of the function RECURSIVE-FFT (excluding the recursive calls) takes $\Theta(n)$ time, where $n$ is the length of the input vector. Therefore, the recurrence for the running time is:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

In other words, we can perform the evaluation of a polynomial of degree-bound $n$ at the complex $n$th roots of unity in time $\Theta(n \log n)$ using the FFT.

### 5.2.4 Interpolation at the complex roots of unity

To complete the polynomial multiplication method we just need to interpolate the complex roots of unity by a polynomial, so that after we can convert the point-value representation back to coefficient representation.

Essentially, we want to compute the inverse DFT, denoted as $\text{DFT}_n^{-1}(y)$. The inverse DFT is given by:

$$a_i = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-ki} \qquad 0 \leq i < n \tag{5.10}$$

If we compare the DFT of Equation 5.8 with Equation 5.10, we see that we can use the FFT (Algorithm 3) to compute both the DFT and its inverse as well. We simply switch the roles of $a$ and $y$, replace $\omega_n$ by $\omega_n^{-1}$, and divide each element of the result by $n$. Thus, we can compute the $\text{DFT}_n^{-1}$ in $\Theta(n \log n)$ time as well.

Having the DFT and its inverse, we can use FFT to transform a polynomial of degree-bound $n$ back and forth between its coefficient and point-value representation in time $\Theta(n \log n)$.

For the polynomial multiplication, we have the *convolution* theorem defined as follows:

**Theorem 5.6** ([CLRS09, Theorem 30.8])**.** *For any two vectors $a$ and $b$ of length $n$, where $n$ is a power of 2,*

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b))$$

*where the vectors $a$ and $b$ are padded with $0$s in order to have length $2n$ and the operator $\cdot$ denotes the component-wise product of two $2n$-element vectors.*

This basically concludes the polynomial multiplication algorithm based on FFT, with $\Theta(n \log n)$ time.

# Chapter 6

# Implementation and experimental results

In the previous chapters we have introduced the two homomorphic MACs (CF1 and CF2) that we intend to implement, as well as the necessary theory behind them. Now we will present some implementation details, and finally an analysis of the efficiency of the two homomorphic MACs.

## 6.1 Programming Language and Libraries

Our choice of programming language was `C`. It is a language that provides more low-level operations which are quite useful to cryptographic applications. However, that can be both an advantage and a disadvantage, as it is easier too create complicated situations with nasty side-effects, which are not clear at first sight. But for our work we think that by using `C` we can obtain the best possible results that otherwise wouldn't be possible in some other higher level languages such as Python or Java.

For cryptographic purposes we always need to work with big numbers, but `C` only has native support for fixed-length arithmetic (normally with machine specific length). Therefore we need to either use an existing big number library, or to implement our own. The latter option is clearly a bad idea since we already have at our disposal many efficient and mature libraries ready for a wide range of machines and architectures.

We choose to use GMP [Gt14] for "bignum" arithmetic. GMP is designed to have good performance with both small numbers and big numbers with thousands of bits. GMP chooses algorithms depending on the size of the operands, so the algorithm used for a small number multiplication is different from the one used to multiply two "bignums". The great speed of GMP is the fruit of many years of optimizations provided by the community, which we can make use of without having to worry about

the underlying machine's hardware. Besides "bignum" arithmetic, GMP also provides the facilities for random numbers generation.

However, GMP does not have support for polynomial arithmetic, which we need for CF tags. Once again, instead of implementing from scratch all the algorithms presented in Chapter 5, we decided to use an established library, in this case NTL [Sho14]. NTL is a high-performance, portable, `C++` library with support for many Number Theory operations, including the FFT multiplication that we need[1].

Lastly, CF makes use of a PRF in both authentication and verifications phases. Just as suggested by Catalano and Fiore, we can use AES since it is extremely efficient. We used Libgcrypt's [Kt14] implementation of AES. Besides cryptographic protocols, Libgcrypt also has support for big number arithmetic, but is based on an older version of GMP, and therefore it is not as efficient for "bignum" operations.

## 6.2 Polynomial arithmetic

We have already seen that (univariate) polynomial arithmetic is critical for the efficiency of the CF schemes. Both addition and multiplication by a constant take $\Theta(n)$ time, where $n$ is the number of coefficients of a polynomial. But on the other hand, the classical multiplication method takes $\Theta(n^2)$ time, and that's why it's better to use the FFT multiplication method described in Chapter 5, which takes $\Theta(n \log n)$ time.

We now present a brief overview on how we implemented these three operations using the libraries mentioned in the previous section.

**Addition** Polynomial addition is quite straightforward: given with two polynomials in coefficient representation (i.e., simple arrays in `C`), addition is simply the addition of two vectors element-by-element. Such addition is done using the function from GMP.

**Multiply by constant** Multiplication by constant is fairly similar to addition, but where one of the arrays has just one element i.e., the constant value. By using the GMP multiplication function, we can take advantage of the possible optimizations present in it where a different algorithm (classical, Karatsuba, FFT, etc) is chosen based on the size of the operands.

**Multiplication of two polynomials** It is not hard to implement a machine specific FFT (i.e., using only machine numbers such as `float`'s), but the complexity involved to achieve an implementation closer to $\Theta(n \log n)$ is very high. One other aspect is that we are working with big numbers, and to

[1] Although it is written in `C++`, we can still link it in such a way that it can be used with our `C` code.

implement FFT with such numbers would introduce even more complexity. The time spent in trying to achieve an efficient implementation would be significant, and we probably would never end up with a time closer to $\Theta(n \log n)$. And also, when implementing a FFT, it is usual to use an iterative version rather than the recursive version of Section 5.2 which improves efficiency, but is also significantly raises the efforts of implementation.

With these limitations in mind, Fateman [Fat10] studied the impact of using libraries like NTL instead of choosing to implement FFT from scratch, and reached the conclusion that it is much better to rely on an existent library given that it is much simpler, and guaranteed to be fast (if the library is obviously efficient). Fateman also noticed that by using NTL for FFT multiplication can be more efficient if NTL is compiled with GMP support[2].

To use the NTL's multiplication function we have to first convert the GMP numbers to the corresponding NTL format, and then use them to create a NTL polynomial. After the NTL multiplication, the inverse conversion is done to obtain the resulting polynomial as GMP coefficients. These conversions obviously introduce a small, non-visible, overhead in the overall multiplication process, but it's only a very small price to pay for a full-fledged FFT multiplication function.

## 6.3 Experimental results

We now present our experimental results by first explaining how we generated our test case circuits, and then by providing more details of the CF's performance. All tests were performed using a x86_64 GNU/Linux on a machine with 4GB RAM and an Intel(R) Core(TM)2 Duo CPU T9600 2.8GHz. The `C` compiler was gcc 4.9.1 x86_64 and the library versions were: GMP 6.0.0, NTL 6.2.1 and Libgcrypt 1.6.2.

**Circuits generation** To measure the performance of CF we tested it with circuits computing polynomials of various degrees. What we did was to create random polynomials using Sage [St14], and then convert them to an appropriate arithmetic circuit representation.

We generated polynomials of degree $n$ between 10 (small enough) and 300 (big enough). The average dimensions of the generated circuits are shown in Table 6.2, and since Pinocchio also contains some example polynomials, we test them with CF as well (we refer to them as Pinocchio's polynomials).

**CF performance** Using the polynomials we previously generated, we can now measure the performance of CF.

[2]NTL's author also advises to compile NTL with GMP support for better results

| $n$ | Inputs | Coeffs | Muls |
|---|---|---|---|
| 10 | 13 | 24 | 128 |
| 20 | 25 | 53 | 545 |
| 50 | 66 | 129 | 3472 |
| 100 | 127 | 307 | 16178 |
| 125 | 149 | 312 | 19766 |
| 150 | 186 | 378 | 29075 |
| 200 | 269 | 532 | 55473 |
| 300 | 372 | 741 | 112195 |

**(a)** Average dimensions of the circuits computing our randomly generated polynomials of degree $n$.

| $n$ | Inputs | Coeffs | Muls |
|---|---|---|---|
| 12 | 4 | 247 | 186 |
| 30 | 6 | 16807 | 55250 |
| 35 | 6 | 32768 | 110624 |
| 40 | 6 | 131022 | 203428 |
| 50 | 6 | 500040 | 350042 |

**(b)** Dimensions of the example circuits of degree $n$ included with Pinocchio.

**Table 6.2:** Dimensions of our randomly generated circuits and from Pinocchio's polynomial examples. Pinocchio's polynomials are of lower degree, but have much more coefficients and multiplications than ours.

The results we are more interested in are the execution times of Eval. In a practical scenario, Eval is executed by the external server, which is expected to have much more computational power than the client who executes Vrfy.

Because Vrfy runs in time proportional to a circuit evaluation, we know that execution times will be similar, with only (small) overhead of Vrfy because of the algebraic operations it performs over the "polynomial tag".

We tested both set of polynomials with CF1 and CF2, but the only difference between the two is that the latter has a slower KeyGen (because of the necessary $D$ exponentiations) and creates succinct tags of constant size (they are simply a group element), which is why we omitted CF2 from the results.

To measure CF's performance, we used a different set of keys for each circuit, and the input messages and labels were chosen at random. The detailed performance results are shown in Table 6.3 (for our randomly generated polynomials) and in Table 6.4 (using Pinocchio's polynomials). For Table 6.3 we run CF once in every one of the 100 circuits of each degree, and the results presented are the average of all these executions; for Table 6.4 we run CF 10 times for each circuit and show the average times for each circuit.

Without much surprise, the authentication algorithm Auth is extremely fast (in less than 2ms we can authenticate about 300 messages). Also note that in a practical scenario, the client might not outsource (and therefore authenticate) all the messages at once, but do it incrementally instead.

Just as expected, the homomorphic evaluation Eval is the slowest operation. However, the obtained results are still very good for a homomorphic primitive. In Table 6.3, to homomorphically evaluate a polynomial of $n = 300$, it "only" takes about 80s. But once again, in a realistic scenario we would be using a much more powerful machine for the homomorphic evaluation and the results would certainly be more efficient.

However, as mentioned by Catalano and Fiore, the tag's size of CF1 grows linearly with the degree of the evaluated circuit, and with $n = 300$ we obtain tags of about 10kB. By using CF2 we can obtain

| $n$ | $\lambda$ | KeyGen (ms) | Auth (ms) | Eval (s) | Vrfy (ms) | Circuit (ms) | Tag Size (kB) |
|---|---|---|---|---|---|---|---|
| 10 | 128 | 0.37 | 0.42 | 0.004 | 0.64 | 0.50 | 0.19 |
| | 256 | 1.23 | 0.63 | 0.007 | 0.62 | 0.58 | 0.38 |
| 20 | 128 | 0.41 | 0.60 | 0.024 | 0.38 | 0.41 | 0.35 |
| | 256 | 1.20 | 0.47 | 0.039 | 0.46 | 0.37 | 0.70 |
| 50 | 128 | 0.39 | 0.44 | 0.31 | 1.17 | 1.06 | 0.83 |
| | 256 | 1.14 | 0.44 | 0.47 | 1.66 | 1.51 | 1.66 |
| 100 | 128 | 0.37 | 0.60 | 2.57 | 6.32 | 5.57 | 1.63 |
| | 256 | 1.16 | 0.79 | 4.01 | 8.61 | 7.44 | 3.26 |
| 125 | 128 | 0.35 | 0.61 | 3.79 | 7.79 | 6.91 | 2.03 |
| | 256 | 1.07 | 0.88 | 5.87 | 10.26 | 8.94 | 4.06 |
| 150 | 128 | 0.48 | 0.71 | 6.94 | 11.63 | 10.31 | 2.43 |
| | 256 | 1.40 | 0.98 | 10.46 | 15.48 | 13.79 | 4.86 |
| 200 | 128 | 0.38 | 0.88 | 17.00 | 23.76 | 22.50 | 3.23 |
| | 256 | 1.22 | 1.32 | 26.40 | 30.87 | 29.08 | 6.49 |
| 300 | 128 | 0.34 | 1.20 | 51.77 | 50.69 | 49.53 | 4.83 |
| | 256 | 1.21 | 1.83 | 79.40 | 64.07 | 62.94 | 9.66 |

**Table 6.3:** CF1 Performance. All circuits representing polynomials of degree $n$ are generated as mentioned before. The security parameter $\lambda$ is either 128 or 256 bits. The Circuit column refers to a circuit evaluation (as demonstrated in Section 3.3) where the inputs are the original messages used in Auth. The Tag Size refers only to CF1, since that for CF2 the tag size is a group element of either 16 or 32 bytes for $\lambda$ of 128 or 256 bits, respectively.

| $n$ | $\lambda$ | KeyGen (ms) | Auth (ms) | Eval (s) | Vrfy (ms) | Circuit (ms) | Tag Size (kB) |
|---|---|---|---|---|---|---|---|
| 12 | 128 | 0.47 | 0.17 | 0.007 | 0.22 | 0.59 | 0.22 |
| | 256 | 1.33 | 0.3 | 0.012 | 0.27 | 0.24 | 0.45 |
| 30 | 128 | 0.72 | 0.36 | 2.89 | 46.75 | 47.63 | 0.51 |
| | 256 | 1.74 | 0.53 | 4.49 | 55.38 | 52.72 | 1.02 |
| 35 | 128 | 0.75 | 0.43 | 6.34 | 100.42 | 97.53 | 0.59 |
| | 256 | 1.85 | 0.56 | 10.72 | 134.26 | 128.12 | 1.18 |
| 40 | 128 | 0.82 | 1.09 | 12.81 | 198.60 | 201.63 | 0.67 |
| | 256 | 2.33 | 0.99 | 19.70 | 227.37 | 229.74 | 1.34 |
| 50 | 128 | 1.44 | 0.44 | 23.92 | 363.63 | 352.24 | 0.83 |
| | 256 | 2.09 | 0.57 | 36.93 | 404.12 | 399.78 | 1.66 |

**Table 6.4:** CF1 Performance. Circuits computing the polynomials used as examples in Pinocchio of degree $n$. For each circuit, the CF was executed $N = 10$ times. The security parameter $\lambda$ is either 128 or 256 bits. The Circuit column refers to a circuit evaluation (as demonstrated in Section 3.3) where the inputs are the original messages used in Auth. The Tag Size refers only to CF1, since that for CF2 the tag size is always one group element of either 16 or 32 bytes for $\lambda$ of 128 or 256 bits, respectively.

smaller tags consisting of only one group $\mathbb{G}$ element, at the cost of losing composition of programs, and by turning the KeyGen operation slower because of the necessary $D$ exponentiations.

It is also not a surprise to see that the Vrfy times do not differ much from the circuit evaluation, with the introduction of only a small overhead responsible for the algebraic computations.

Regarding the results of Pinocchio's polynomials in Table 6.4, their circuit evaluation consists of a much larger number of gate multiplications than our randomly generated polynomials. Because of that, their circuit evaluation takes more time which makes both the homomorphic evaluation Eval and Vrfy slower.

The full source code and the respective documentation is available online as a Git repository[3].

[3] `https://bitbucket.org/ntfc/cf-homomorphic-mac/overview`

# Chapter 7

# Conclusion

We now present a comprehensive synthesis of all the work accomplished, as well as what could be done in the future in order to further enhance this work.

Once the objectives of this dissertation were defined, we started studying some interesting primitives regarding delegation of computations. We did an informal review of Homomorphic Encryption, Verifiable Computation and Succinct Non-Interactive Arguments of Knowledge, as well of Homomorphic Authentication.

All of these primitives need some way to represent the computations so that both clients and servers are able to execute or verify them. The common way of doing it is by using a circuit representation of the program – boolean or arithmetic. We also studied two tools that are able to convert code from higher level languages to an equivalent circuit representation.

We then proceeded to make a more deep analysis of Homomorphic Authentication, specially Homomorphic Signatures and Homomorphic MACs. We were able to list and understand their current limitations in both security and practicability, which lead us to the conclusion that Homomorphic Signatures are not yet practical nor totally secure.

We then chose to implement the two constructions of Catalano and Fiore [CF13] because they seemed to be ones with more practical application. Our main focus was on determining if the homomorphic computation of a function (to obtain a MAC) was practical so that it could easily be computed by a more powerful external machine.

In order to achieve efficient results, we had to study an efficient method for the multiplication of two polynomials in $\Theta(n \log n)$ time: the Fast Fourier Transform. This faster method becomes specially critical when the degree of the polynomials is high.

Finally, after having an implementation of Catalano and Fiore [CF13] in C, and by using the FFT for polynomial multiplications, we were ready to measure its performance by using random polynomials, described as arithmetic circuits.

As we stated in the previous chapter, the obtained results were very positive, which means that Catalano and Fiore [CF13] is practical. The major problem is within the homomorphic computation as expected, but if we consider that these computations are supposed to be executed by a powerful machine, our results seem very good.

## 7.1 Future work

From a practical point of view, there are many possible things that could be done in the future. The first and the most obvious one is to implement the work of Backes, Fiore, and Reischuk [BFR13], which we could use to obtain an efficient verification algorithm (in the amortized sense).

One other area with room for improvement is the size of the input messages we use, which now is only of 128 and 256 bits. If we supported bigger message sizes we would be able to use more practical message values. A possible solution would be the use of a Luby and Rackoff [LR88] construction (using AES as the round function) that would allows us to use messages up to 512 bits.

Another interesting work would be the creation of a library with support for various Homomorphic Authentication primitives, much like the efforts done with HELib[1] for Homomorphic Encryption. Such library could also have the possibility of a better communication with Pinocchio or TinyRAM to easily convert C programs into arithmetic circuits.

[1] https://github.com/shaih/HElib

# References


[Adi08] Ben Adida. "Helios: Web-based Open-audit Voting". In: *Proceedings of the 17th Conference on Security Symposium*. SS'08. San Jose, CA: USENIX Association, 2008, pp. 335–348. URL: http://dl.acm.org/citation.cfm?id=1496711.1496734.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. "Short Group Signatures". In: *Advances in Cryptology - CRYPTO 2004*. Ed. by Matt Franklin. Vol. 3152. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pp. 227–242. ISBN: 978-3-540-22668-0. DOI: 10.1007/978-3-540-28628-8_3.

[BCCT11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Report 2011/443. 2011. URL: http://eprint.iacr.org/.

[BCGTV13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge*. Cryptology ePrint Archive, Report 2013/507. http://eprint.iacr.org/. 2013.

[BF11] Dan Boneh and David Mandell Freeman. "Homomorphic Signatures for Polynomial Functions". In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by KennethG. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 149–168. ISBN: 978-3-642-20464-7. DOI: 10.1007/978-3-642-20465-4_10. URL: http://dx.doi.org/10.1007/978-3-642-20465-4_10.

[BFR13] Michael Backes, Dario Fiore, and Raphael M. Reischuk. "Verifiable delegation of computation on outsourced data". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. CCS '13. Berlin, Germany: ACM, 2013, pp. 863–874. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516681. URL: http://doi.acm.org/10.1145/2508859.2516681.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. "Evaluating 2-DNF Formulas on Ciphertexts". In: *Proceedings of the Second International Conference on Theory of Cryptography*. TCC'05. Cambridge, MA: Springer-Verlag, 2005, pp. 325–341. ISBN: 3-540-24573-1, 978-3-540-24573-5. DOI: 10.1007/978-3-540-30576-7_18. URL: http://dx.doi.org/10.1007/978-3-540-30576-7_18.

[CF13] Dario Catalano and Dario Fiore. "Practical Homomorphic MACs for Arithmetic Circuits". In: *Advances in Cryptology - Eurocrypt 2013*. Vol. 7881. 2013, p. 336. DOI: 10.1007/978-3-642-38348-9_21.

[CFGN14] Dario Catalano, Dario Fiore, Rosario Gennaro, and Luca Nizzardo. "Generalizing Homomorphic MACs for Arithmetic Circuits". English. In: *Public-Key Cryptography – PKC 2014*. Ed. by Hugo Krawczyk. Vol. 8383. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 538–555. ISBN: 978-3-642-54630-3. DOI: 10.1007/978-3-642-54631-0_31. URL: http://dx.doi.org/10.1007/978-3-642-54631-0_31.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. "A Secure and Optimally Efficient Multi-authority Election Scheme". In: *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'97. Konstanz, Germany: Springer-Verlag, 1997, pp. 103–118. ISBN: 3-540-62975-0. URL: http://dl.acm.org/citation.cfm?id=1754542.1754554.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[Elg85] Taher Elgamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *Information Theory, IEEE Transactions on* 31.4 (July 1985), pp. 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074. URL: http://dx.doi.org/10.1109/TIT.1985.1057074.

[Fat10] Richard J. Fateman. "Can You Save Time in Multiplying Polynomials By Encoding Them as Integers?" In: (2010). URL: http://www.cs.berkeley.edu/~fateman/papers/polysbyGMP2.pdf.

[Gen09] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 169–178. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536440. URL: http://doi.acm.org/10.1145/1536414.1536440.

[GGP09] Rosario Gennaro, Craig Gentry, and Bryan Parno. "Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers". In: (2009). URL: https://eprint.iacr.org/2009/547.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for Hard Lattices and New Cryptographic Constructions". In: *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*. STOC '08. New York, NY, USA: ACM, 2008, pp. 197–206. ISBN: 978-1-60558-047-0. DOI: 10.1145/1374376.1374407. URL: http://doi.acm.org/10.1145/1374376.1374407.

[Gt14] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. 6.0.0. 2014. URL: https://gmplib.org.

[GW12] Rosario Gennaro and Daniel Wichs. "Fully Homomorphic Message Authenticators". In: *IACR Cryptology ePrint Archive* 2012 (2012), p. 290. URL: http://eprint.iacr.org/2012/290.

[JMSW02] Robert Johnson, David Molnar, Dawn Song, and David Wagner. "Homomorphic Signature Schemes". In: *Topics in Cryptology — CT-RSA 2002* 28913.Section 4 (2002), pp. 244–262. DOI: 10.1007/3-540-45760-7_17.

[Kt14] Werner Koch and the Libgcrypt development team. *Libgcrypt*. 1.6.2. 2014. URL: https://www.gnu.org/software/libgcrypt/.

[LR88] Michael Luby and Charles Rackoff. "How to construct pseudorandom permutations from pseudorandom functions". In: *SIAM Journal on Computing* 17.2 (1988), pp. 373–386.

[Pai99] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-65889-4. DOI: 10.1007/3-540-48910-X_16. URL: http://dx.doi.org/10.1007/3-540-48910-X_16.

[PF79] Nicholas Pippenger and Michael J. Fischer. "Relations Among Complexity Measures". In: *J. ACM* 26.2 (Apr. 1979), pp. 361–381. ISSN: 0004-5411. DOI: 10.1145/322123.322138. URL: http://doi.acm.org/10.1145/322123.322138.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. "Pinocchio: Nearly Practical Verifiable Computation". In: *Proceedings of the 2013 IEEE Symposium on Security*

*and Privacy*. SP '13. Source available at https://vc.codeplex.com/. Washington, DC, USA: IEEE Computer Society, 2013, pp. 238–252. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.47. URL: http://dx.doi.org/10.1109/SP.2013.47.

[RAD78] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation* 32.4 (1978), pp. 169–178.

[RMCR01] Ronald L. Rivest, Silvio Micali, Suresh Chari, and Tal Rabin. "Two Signature Schemes". In: Presented at Cambridge University, 2001.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342.

[Sho14] Victor Shoup. *NTL: A Library for doing Number Theory*. 6.2.1. 2014. URL: http://www.shoup.net/ntl/.

[St14] William Stein and the Sage development team. *Sage*. 6,3. 2014. URL: http://www.sagemath.org/.

[SY10] Amir Shpilka and Amir Yehudayoff. "Arithmetic Circuits: A Survey of Recent Results and Open Questions". In: *Foundations and Trends in Theoretical Computer Science* 5 (Mar. 2010), pp. 207–388. ISSN: 1551-305X. DOI: 10.1561/0400000039. URL: http://dx.doi.org/10.1561/0400000039.

[Tar72] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[Tur37] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230. URL: http://plms.oxfordjournals.org/content/s2-42/1/230.short.