Karlsruhe Institute of Technology

Efficient Secure Function Evaluation using Garbled Arithmetic Circuits and Untrusted Tamper–Proof Hardware

Diplomarbeit

Johannes Weiß

Institut für Kryptographie und Sicherheit
Europäisches Institut für Systemsicherheit
Fakultät für Informatik
Karlsruher Institut für Technologie

Betreuender Professor:	Prof. Dr. rer. nat. Jörn Müller–Quade
Betreuender Mitarbeiter:	Dipl.-Inform. Daniel Kraschewski

Bearbeitungszeit: 1. Juli 2012 – 31. Dezember 2012

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 18. Dezember 2012

Johannes Weiß

Efficient Secure Function Evaluation using Garbled Arithmetic Circuits and Untrusted Tamper–Proof Hardware

Johannes Weiß <diplomarbeit@johannesweiss.eu>

Typeset using LaTeX on January 16, 2013, 15:52.

Repository Hashes: tree: fbb5261ed844f70f7e1bb5e4b84c3851cc8ac477, last commit: 37436f53965c2c493c173a0f025c549cc806f8af

```
M bibliography.bib
M conclusion.tex
M introduction.tex
M security.tex
```

Abstract

Today, strong cryptography plays a very important role. Cryptography is of crucial importance mainly, but not limited, to important transactions via the Internet and other communication networks because the information should reach its destination reliably, confidentially and with integrity. The telematics research solves the reliability problem, but cryptography is used to ensure confidentiality and integrity. Since the overall problem is very complex, an increasing effort to use cryptographic primitives and protocols is discoverable. The benefit of this component based architecture is the possibility to prove the security of the components individually. Of course, the proofs must take into account the composability of the components which enables to build complex and secure systems from smaller building blocks.

This thesis concerns itself with cryptographic primitives for *Secure Multi–Party Computations* (MPC). MPC are joint computations of a set of parties which reveal the result of the computation to any party and nothing else. Every party should only learn information that can be calculated from its own input and the result. This thesis mainly deals with *Oblivious Polynomial Evaluation* (OPE), a subset of general MPC. OPE allows two parties to jointly evaluate a polynomial where the first party chooses the polynomial and learns nothing. The second party chooses the node and only learns the evaluated result of the polynomial at the chosen node. In addition to the evaluation of polynomials which is an obvious use case of OPE, OPE has many interesting applications, such as the share generation for *Shamir's Secret Sharing*. The methodology of this thesis also applies to the larger class of *Secure Function Evaluation* (SFE) which enables to securely evaluate arbitrary functions.

This thesis examines various approaches, leading to the novel result of a cryptographic protocol realizing OPE in linear time. The security of the methodology is proved in the *Universal Composability* (UC) framework which places very strict demands on the security of cryptographic protocols.

Alongside the theoretical debate, this thesis also features an efficient, secure and working implementation which manifests the properties of the protocol mentioned above.

Zusammenfassung

In der heutigen Zeit ist starke Kryptographie nicht mehr wegzudenken. Vor allem, aber nicht ausschließlich, bei wichtigen Transaktionen über das Internet oder andere Kommunikationsnetze ist es von herausragender Bedeutung, dass Informationen zuverlässig, integer und vertraulich das vorgesehene Ziel erreichen. Die Zuverlässigkeit wird von der Telematik untersucht. Die Integrität und die Vertraulichkeit werden jedoch mit kryptographischen Methoden sichergestellt. Da die Gesamtheit ein sehr komplexes Problem ist, geht man zunehmend dazu über komplexe Verfahren auf kryptographischen Primitiven und Protokollen aufzubauen, die bewiesen sicher und komponierbar sind. Wie in anderen Wissenschaften können die einzelnen Komponenten dann zu einem funktionierenden Gesamtsystem zusammengesteckt werden.

Die vorliegende Arbeit beschäftigt sich mit kryptographischen Primitiven zur sicheren Mehrparteienberechnung. Sichere Mehrparteienberechnungen sind Berechnungen, die von mehreren Parteien durchgeführt werden, wobei keine Partei mehr Wissen erlangt als sie aus ihrer Eingabe und der Ausgabe der Berechnung ohnehin hätte berechnen können. Diese Arbeit beschäftigt sich hauptsächlich mit der Primitive *Oblivious Polynomial Evaluation* (OPE), die eine Unterklasse der sicheren Mehrparteienberechnung darstellt. Mit OPE können zwei Parteien gemeinsam ein Polynom auswerten, wobei die erste Partei das Polynom festlegt und keine Informationen erhält. Die zweite Partei soll das Polynom an einer Stützstelle auswerten können und das Ergebnis, jedoch nicht das Polynom an sich, erhalten. Darüber hinausgehende Informationen können weder die erste, noch die zweite Partei errechnen. Neben der selbstverständlichen Anwendung von OPE zum Evaluieren von Polynomen gibt es für die kryptographische Primitive OPE noch zahlreiche weitere Anwendungen, zum Beispiel die Generierung der *Shares* bei *Shamir's Secret Sharing*. Die in der Arbeit entwickelten Methoden lassen sich auch auf die größere Klasse *Secure Function Evaluation* (SFE), mit der allgemeine Funktionen ausgewertet werden können, anwenden, der Fokus liegt jedoch auf OPE.

Um dieses ambitionierte Ziel zu erreichen untersucht diese Arbeit verschiedene Ansätze und kommt zu einem bemerkenswerten Ergebnis: Mit den in dieser Arbeit vorgestellten Methoden ist es möglich beliebige Polynome mittels OPE über großen endlichen Körpern in Linearzeit auszuwerten. Die Sicherheit des Verfahrens wird unter dem strengen Sicherheitsbegriff der *Universal Composability* (UC) bewiesen. Neben der theoretischen Ausarbeitung liegt dieser Thesis auch eine vollständige Implementierung bei, die die angesprochenen Eigenschaften zeigt.

Contents

1 Introduction

A *Secure Multi–Party Computation* (MPC) or *Secure Function Evaluation* (SFE) is the joint calculation of an arbitrary function f with private inputs from a set of mutually distrusting parties. If the size of the set of parties for an MPC is two, the computation is also called a two–party computation. The area of research of secure multi–party computations was founded by Yao [Yao82]. His famous *Millionaires' Problem* discusses two millionaires, who are interested in knowing which of them is richer without revealing the actual values of their wealth. The problem is evaluating the function $f(a, b) = a \geq b$, where the first millionaire inputs the value of his wealth as a and the second inputs b. The result of the evaluation is either *false* if the second millionaire is richer or *true* meaning the opposite. From the result and the own input, neither party is able to calculate the wealth of the other millionaire. Another example for real–world multi–party computations are democratic elections. The eligible voters input their private vote and the result of the calculation is the percentage breakdown.

Oblivious Polynomial Evaluation (OPE) [NP99, NP06] is a special case of SFE. In contrast to arbitrary functions that are known by the involved parties, OPE only allows the first party to privately submit a polynomial p and another party to evaluate the polynomial at one node x. The second party obtains the result $p(x)$ but not the polynomial, the first party learns nothing. A secure and efficient OPE protocol is a useful and interesting primitive because many other cryptographic problems can be based on OPE, for example the share generation for *Shamir's Secret Sharing* [Sha79].

Oblivious Affine Function Evaluation (OAFE) [DKMQ12] is a primitive that allows to obliviously evaluate affine functions. The David & Goliath protocol [DKMQ12] presents a secure and efficient OAFE implementation based on only one stateful tamper–proof hardware token. Based on OAFE, this thesis works out the novel result of OPE in linear time which is asymptotically as fast as usual polynomial evaluation. Alongside the theoretical methodology, this thesis features an efficient, secure, and working implementation of the proposed protocol. The presented approaches are also suitable for general SFE (including *Square & Multiply* [Knu81]), however the security is only proved for OPE. The security of the protocol against passive and active adversaries is information–theoretically proved in the *Universal Composability* (UC) framework [Can01].

1.1 Related Work

The notion of *universal composability* in the UC framework by Canetti [Can01] places strict demands on the security of cryptographic protocols. Proofs in the UC framework have to show that any environment is unable to distinguish between an ideal functionality and the implemented protocol, even when malicious adversaries misuse the protocol in some unpredictable way. The benefit of the strict requirements of this notion is that UC–secure protocols can be composed and ran concurrently while still staying safe without

the necessity of additional security proofs. This thesis uses universal composability as the notion of security.

Yao defined the problem of multi–party computations and his *Garbled Circuit* (GC) approach [Yao86] describes encrypted evaluation of (boolean) circuits. But despite the versatility of Yao's garbled circuits, the approach is not suitable for arithmetic functions on large fields. Because the GC approach requires embedded truth tables in every gate, an adoption to larger finite fields would have the consequence of a quadratic blowup of every truth table in every gate (cf. [NPS99]). This thesis solves these problem efficiently for arbitrary polynomials.

Besides the GC approach, this thesis is also related to *Efficient Multi-Party Computation Over Rings* by Cramer et al. [CFIK03]. The difference is that Cramer's approach is based on the simulation of formulas by bounded–width programs by Cleve [Cle91] which does not support *Square & Multiply* [Knu81] and the complexity is polynomial (with a low degree) in the number of arithmetic operations performed. The writings of Naor et al. [NP99, NP06] also describe OPE but have polynomial complexity and are based on *Oblivious Transfer* (OT) [Rab81].

How to Garble Arithmetic Circuits by Applebaum et al. [AIK12] describes the garbled evaluation of arithmetic circuits. Many ideas used in this thesis are inspired by Applebaum et al. The difference between Applebaum's paper and this thesis, is that the former relies on computational assumptions to be secure. In contrast, this thesis does not rely on computational assumptions and proves the methodology to be information–theoretically secure. This thesis adapts some of Applebaum's ideas to work with *Oblivious Affine Function Evaluation* (OAFE) [DKMQ12].

1.2 Acknowledgments

I wish to thank my advisor Daniel Kraschewski who supported me throughout the research and writing of this thesis. He donated a large amount of his limited time for discussions and explorations crucial for my thesis and allowed this thesis to be my own work but steered me in the right direction when needed. I am also thankful for his patience introducing me to this exciting topic.

1.3 Outline

This thesis starts by introducing the reader to the general methodology, which is discussed in depth thereafter (Chapter 2). Next, the security of the methodology is analyzed and proved in the UC framework (Chapter 3). Since this thesis also implements the proposed protocol, the implementation is covered in Chapter 4. An evaluation of both, the methodology and the implementation (Chapter 5) followed by the conclusion (Chapter 6) rounds out this thesis. And since a large amount of the time taken for this thesis was affected by researching, a brief tour of the discontinued approaches is given to the reader (Chapter 7).

2 Methodology

This chapter forms a major part of this thesis and gives insight into the mathematical foundations used in this thesis. It starts by describing the overall process followed by an in–depth description of the individual components.

2.1 Illustration

This section informally illustrates the key ideas and methods proposed by this thesis. The following sections contain a more formal bottom–up description that is rather cumbersome to understand without grasping the overall picture.

The goal is to obliviously evaluate a polynomial by two mutual distrusting parties. Informally, that means to evaluate a polynomial ($f(x) = \sum_i a_i x^i$) where the first party chooses f and the second party chooses x. It is crucial that neither party learns the choice of the other party. The naive and simple solution is to instruct a trusted third party to evaluate the polynomial—provided by the first party—at the node x provided by the second party. Of course, this solution is just as simple as insecure because the third party could reveal the secrets at any time.

This thesis expresses the general polynomial as affine functions that evaluate parts of the complete function. The entirety of the affine functions exactly represents the original polynomial. The process is based on dividing the input polynomial into affine functions because *Oblivious Affine Function Evaluation* (OAFE) [DKMQ12] is the main building block. Every OAFE enables one party to choose an affine function that another party can evaluate obliviously. The OAFE guarantees that the second party does not learn the affine function and that the first party does not learn the node at which the second party evaluated the function. Additionally, OAFE guarantees that every linear function can only be evaluated once. To maintain the security property for the polynomial case, neither party should be able to learn anything from intermediate results.

The role both parties play differs heavily: The first party—usually named Goliath—is in possession of the polynomial to evaluate and sets up the OAFEs. The second party—usually named David—is less powerful: It does only evaluate entities issued by Goliath and uses the already configured OAFEs. But in the end, only David obtains the result. Since the OAFEs play an important role, it is crucial to envision that every OAFE Goliath generates can be evaluated by David only once. In other words, an OAFE enables some party to evaluate an affine function at one node and guarantees that no further evaluations are possible. To prevent a corrupted Goliath from intercepting information when David evaluates one of the OAFEs, Goliath is forced to commit itself to a finite number of OAFEs a priori. Goliath is also forced to commit itself to all parameters for all OAFEs. The OAFE implementation from Döttling, Kraschewski and Müller–Quade [DKMQ12] realizes the OAFE functionality with exactly one tamper–proof hardware token which is physically sent to the other party. Because the OAFE functionality realizes all OAFEs

together, Goliath needs to commit itself to the OAFE configuration at the beginning. Goliath configures all OAFEs before David even starts the process.

All following steps will be performed by David alone, no further interaction with Goliath is necessary. An honest David acts completely deterministically and will receive the function's final evaluated value $f(x)$.

In the following, the process of dividing the function in smaller parts is examined. The term *(arithmetic) circuit* meaning a rooted evaluation tree (e.g., Figure 2.1) of a function is used thereafter. The the leaf nodes of the evaluation tree contain values or inputs and the inner nodes—called *(arithmetic) gates*—perform arithmetic operations.

The approach pursued here is to divide the arithmetic circuit that equals the overall function to smaller sub–circuits. Each sub–circuit can be composed of arithmetic gates and other sub–circuits. David is commissioned to evaluate each sub–circuit to an intermediate value. David then stores all intermediate values since they are used to compute the following sub–circuits. Whenever one sub–circuit A contains a node that is another sub–circuit B, this node gets replaced by the intermediate output value of B before evaluating A. Obviously, the intermediate values are a potential security threat: A passively corrupted David could try to derive information from these intermediate results. An actively corrupted David could additionally try to modify the intermediate values and gain supplemental information out of that. To not directly reveal the value of an evaluated sub–circuit, the intermediate values have to be *encrypted*. To be able to identify an illicit forgery, the overall evaluation has to fail if an attacker tries to illicitly modify an intermediate result. In this thesis, the intermediate values are tuples of two values. The decryption of both values represents—when well–formed—the same value. The gain of this technique is that an attacker cannot change the value and preserve the well–formedness without knowing the encryption keys. Any change to one (or both) tuple values causes the intermediate value to be destroyed (formally represented as $\perp$). After having evaluated the last sub–circuit, its resulting encrypted tuple undergoes a special procedure revealing the unencrypted scalar result value. Details and the formal description of the procedures mentioned above can be found in the following sections.

The overall process is very straightforward: Goliath analyzes the polynomial and transforms it into a circuit (the entire Figure 2.1). This circuit is then divided to sub–circuits (the boxes bounded by dashed lines). All the sub–circuits are then encoded in a way that David cannot learn anything private to Goliath. To achieve that, Goliath sets up OAFEs and hands the OAFEs along with a description of the sub–circuits to David. After sending the circuits and the OAFEs, Goliath can shut down because it is no longer needed in the process. As soon as David receives the sub–circuit description and the OAFEs, it starts the evaluation, one sub–circuit after the other. To discuss the evaluation, the sample polynomial $1 + 2x + 3x^2$ (see Figure 2.1) is used hereinafter. In the whole procedure, David should not learn the coefficients (a_0, a_1 and a_2) of the polynomial $a_0 + a_1x + a_2x^2$ but is able to learn the degree of the polynomial. The first step for David is to evaluate the sub–circuit only containing x. This is in fact only one sub–circuit that appears three times in the figure. The next step would be to evaluate the sub–circuit only encoding 3 (remember, not learning the value 3!), then the sub–circuit encoding $x^2 (= x \cdot x)$ and finally the overall circuit that encodes $1 + 2x + 3x^2$. Usually and as in the example above, the sub–circuits are of maximal size. Notably, a sub–circuit can only contain multiplications that exclusively depend on either constants or stored intermediate values.

To simplify the analysis of the different components and to show the modularity, special

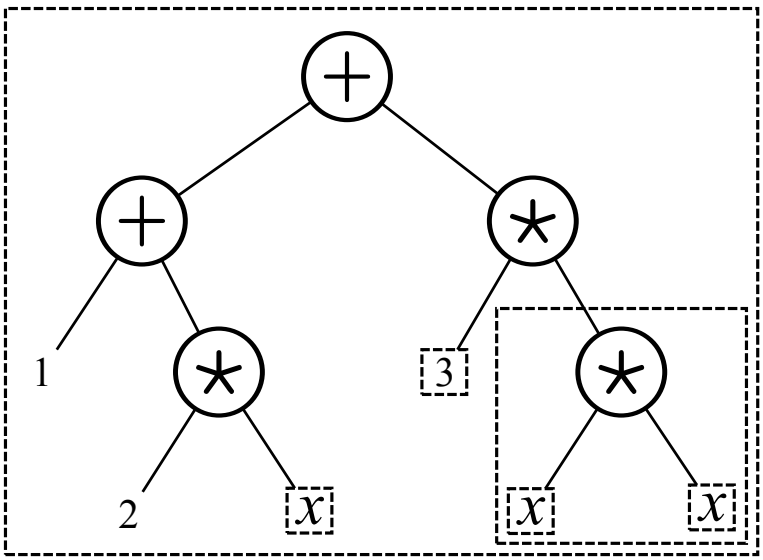

Figure 2.1: Circuit encoding the polynomial $1 + 2x + 3x^2$ including its (sub–)circuits (dashed boxes)

terms are used throughout the entire thesis. A mapping from the informal terms used above and the terms used otherwise (and defined in the following sections) follows.

In the informal illustration above, the encoded tuples play an important role. They represent values and the sub–circuits evaluate to these encrypted tuples. In fact, the sub–circuits are also twofold, they have a left and a right side that eventually evaluate to two values: Both sides of the encrypted tuple. For technical reasons, the two sides of the circuits and the tuples are often handled apart. They still represent the same value encrypted with different keys. For this reason the terms below all exist in two forms, the *Double* form (such as *Double Randomized Affine Encoding*) and the regular form (such as *Randomized Affine Encoding*). At some point in the processing, they are split syntactically, two regular entities are just the same as one Double entity. The difference is mainly the same as naming tuple $\widetilde{a} = (\widetilde{a_l}, \widetilde{a_r})$ *Double Value* and $\widetilde{a_l}$ and $\widetilde{a_r}$ *Value*s.

- The intermediate values that are encrypted twofold are named *Double Randomized Affine Values* (DRAV), defined in Section 2.3.

- The encodings of the sub–circuits are named *Double Affine Randomized Encodings* (DRAE) and *Randomized Affine Encodings* (RAE), defined in Section 2.4. Each (D)RAE is assigned to an intermediate variable that occurs in the subsequent (D)RAEs. A (D)RAE is a specification of calculations that have to be performed until obtaining the (D)RAEs result value.

- The overall circuit representing exactly one function is named *Double Affine Randomized Circuit* (DRAC) or *Affine Randomized Circuit* (RAC) defined in Section 2.5. A DRAC is the collectivity of DRAEs each assigned to intermediate variables storing their result values.

2.2 Definitions

This section defines important entities that are used in the following sections.

2.2.1 Field $\mathbb{K}$

Throughout this thesis, $\mathbb{K}$ represents an arbitrary large finite field, such as $\mathbb{F}_{2^{256}}$. When not mentioned otherwise, this thesis assumes the field $\mathbb{F}_{2^{256}}$ specified by the irreducible polynomial $1 + x^2 + x^5 + x^{10} + x^{256}$.

2.2.2 Oblivious Affine Function Evaluation

The main building block and the only external cryptographic primitive of this thesis are *Oblivious Affine Function Evaluations* (OAFEs). More precisely, *Sequential one–time OAFE* as defined by Döttling, Kraschewski and Müller–Quade [DKMQ12]. The OAFE functionality enables to sequentially evaluate n affine functions specified a priori. The OAFE functionality is implemented by the David & Goliath protocol [DKMQ12] and exactly one stateful tamper–proof hardware token. In other words, one party can specify n affine functions and another party will be able to sequentially evaluate the functions exactly once. The evaluating party will only learn one evaluation of each function f_i but not their definition. After all n functions have been evaluated the tamper–proof hardware token can be physically destroyed.

$$
\begin{aligned}
i &\in \{1, \ldots, n\} \\
f_i(x_i) &= a_i x_i + b_i \\
&= \begin{pmatrix} a_{i,1} \\ a_{i,2} \\ a_{i,3} \\ \vdots \end{pmatrix} x_i + \begin{pmatrix} b_{i,1} \\ b_{i,2} \\ b_{i,3} \\ \vdots \end{pmatrix}
\end{aligned}
$$

For the sake of simplicity, the term *OAFE* is used slightly different in this thesis: The term *OAFE* is a functionality to evaluate one affine function. The plural (*OAFEs*) means a set of OAFEs that enables to evaluate multiple affine functions. Therefore the collectivity of OAFEs as defined in this thesis can be implemented by one call to the *Sequential one–time OAFE* protocol as defined by Döttling, Kraschewski and Müller–Quade [DKMQ12].

2.3 Dual Randomized Affine Values

Dual Randomized Affine Values (DRAVs) are encrypted, signed values representing a scalar value of the field $\mathbb{K}$. Each DRAV is encrypted with a pair of *dual keys*. The first dual key, the *static key*, remains the same in the whole DRAC (see Section 2.5) generation procedure and is usually denoted as (α_l, α_r). There are special requirements on the nature of static keys: α_l, α_r, and $\alpha_l + \alpha_r$ have to be invertible (for the reason, see Sections 2.3.3 and 2.4.2.2). The second dual key, the *dynamic key* is a short–lived key, usually represented as (β, β'). Two DRAVs only share the same dynamic key by chance but always share the same static key. Encoding a regular value $v \in \mathbb{K}$ as a DRAV

$\widetilde{v}$ is straightforward: For the sake of simplicity this thesis assumes a function $E(v)$ (also named *universal encoding function*) that is implicitly parametrized by the static dual key (α_l, α_r) and is able to generate a new dynamic dual key (β, β') uniformly at random. Just like $E(v)$, this thesis also assumes $D(\widetilde{v})$, named the *universal decoding function*. The universal decoding function is assumed to be implicitly parametrized by the encryption keys from the context it is used in. Both functions are defined in the following.

$$\begin{aligned} \hat{E}(v; \alpha_l, \alpha_r, \beta, \beta') &= (\alpha_l \cdot v + \beta, \alpha_r \cdot v + \beta') \\ E(v) &= \hat{E}(v; \alpha_l, \alpha_r, \beta, \beta') \qquad \text{(implicitly knowing the keys)} \end{aligned} \tag{2.1}$$

The values β and β' are fresh values uniformly at random. In finite fields, β and β' therefore one–time encrypt both tuple components, as such a DRAV is one–time pad encrypted. α_l and α_r are also values uniformly at random but have—in contrast to β and β'–the same value in all DRAVs throughout the whole process.

Notationally, the DRAV encoding a scalar value v is usually denoted as $\widetilde{v}$. $\widetilde{v}$'s tuple components are usually called $\widetilde{v_l}$ and $\widetilde{v_r}$: $(\widetilde{v_l}, \widetilde{v_r}) = \widetilde{v} = E(v)$. Where appropriate, DRAVs are also denoted in a vector notation $\begin{pmatrix} \widetilde{v_l} \\ \widetilde{v_r} \end{pmatrix} \hat{=} (\widetilde{v_l}, \widetilde{v_r}) = E(v)$. The universal decoding function $D(\widetilde{v})$ is:

$$\begin{aligned} \hat{D}(\widetilde{v}; \alpha_l, \alpha_r, \beta, \beta') &= \begin{pmatrix} \frac{\widetilde{v_l} - \beta}{\alpha_l} \\ \frac{\widetilde{v_r} - \beta'}{\alpha_r} \end{pmatrix} \\ \begin{pmatrix} D_l(\widetilde{v}) \\ D_r(\widetilde{v}) \end{pmatrix} &= \hat{D}(\widetilde{v}; \alpha_l, \alpha_r, \beta, \beta') \qquad \text{(implicitly knowing the keys)} \\ D(\widetilde{v}) &= \begin{cases} D_l(\widetilde{v}) & \text{if } D_l(\widetilde{v}) = D_r(\widetilde{v}) \\ \perp & \text{otherwise } (\widetilde{v} \text{ is non–well–formed}) \end{cases} \end{aligned} \tag{2.2}$$

In equation 2.2 one can see that the universal decoding function D assumes context–specific dual keys (α_l, α_r) and (β, β'). If an attacker at some point passes a DRAV that is not expected to be decoded at this point, the decoding will fail and evaluate to $\perp$. This is evident because D will decode using the dual keys of the DRAV that was expected to be decoded (see also Lemma 2).

The universal encoding and decoding functions only serve the purpose of describing the methodology. In the reality, especially the function D is never called by anyone. That would not even be possible since only the party in possession of the keys (Goliath) has the information to do so but because of the non–interactivity of the protocol, Goliath is not active anymore when the second party (not in possession of the keys, usually named David) starts the evaluation. The real protocol encodes the encryption and decryption into the OAFEs in a way that as soon as David tries to cheat everything eventually becomes uniform randomness. To cheat means to disrespect the protocol description and to misbehave in any way. The property to turn everything to uniform randomness as soon as a party tries to cheat is done using carefully chosen linear functions realized as OAFEs. The setup of these linear functions assures the addition of uncorrelated uniform randomness as soon as David tries to cheat. One can envision the process as applying a one–time pad encryption that is only correctly removed if the other party did not try to

cheat. As soon as the second party tries to cheat this resembles the attempt to undo a one–time pad encryption using another one–time pad. This clearly leads to a one–time pad encryption which can only be removed using another unknown one–time pad.

Lemma 1. *A party that is not in possession of the encryption (dual) keys will not learn anything from a DRAV. A DRAV is a tuple consisting of two values uniformly at random that is uncorrelated to any other DRAV.*

Proof. In a finite field, the addition of some value r uniformly at random to another value $x \in \mathbb{K}$ is uniformly at random. Therefore $\alpha_l \cdot v + \beta$ and $\alpha_r \cdot v + \beta'$ are two values uniformly at random if and only if β and β' are uniformly at random. Per definition, every time a DRAV gets encoded, β and β' are fresh uniformly distributed random values. Trivially, a DRAV is therefore uncorrelated to any other DRAV.

□

2.3.1 Well–Formed DRAVs

A DRAV $\widetilde{v}$ is well–formed iff $D(\widetilde{v}) \neq \perp \Leftrightarrow D_l(\widetilde{v}) = D_r(\widetilde{v})$. Therefore, if $D(\widetilde{\chi}) = \perp$, χ is non–well–formed. Obviously the well–formedness of a DRAV depends on the encryption keys used to decrypt. Decoding well–formed DRAVs using any other decoding function will in all probability lead to $\perp$ because the well–formed DRAV looks non–well–formed to the other decoding function.

Lemma 2. *DRAV well–formedness is a function of the keys used to decrypt it.*

Proof. Assuming two well–formed DRAVs $\widetilde{a}$ and $\widetilde{b}$. Both DRAVs have their respective decoding function D^a and D^b and their dynamic dual key (say (β_1, β_2) for $\widetilde{a}$ and (β_3, β_4) for $\widetilde{b}$). The following equations show the decoding of the DRAVs $\widetilde{a}$ and $\widetilde{b}$ with the correct keys and the wrong keys.

$$
\begin{aligned}
D^a(\widetilde{a}) &= D(\widetilde{a}) \qquad \text{(using } \widetilde{a}\text{'s keys)} \\
D^b(\widetilde{b}) &= D(\widetilde{b}) \qquad \text{(using } \widetilde{b}\text{'s keys)} \\
a = D^a(\widetilde{a}) &\triangleq \begin{pmatrix} \frac{(\alpha_l \cdot a + \beta_1) - \beta_1}{\alpha_l} \\ \frac{(\alpha_r \cdot a + \beta_2) - \beta_2}{\alpha_r} \end{pmatrix} = a \\
a' = D^a(\widetilde{b}') &\triangleq \begin{pmatrix} \frac{(\alpha_l \cdot b + \beta_3) - \beta_1}{\alpha_l} \\ \frac{(\alpha_r \cdot b + \beta_4) - \beta_2}{\alpha_r} \end{pmatrix} \triangleq \begin{pmatrix} b + \frac{\beta_3 - \beta_1}{\alpha_l} \\ b + \frac{\beta_4 - \beta_2}{\alpha_r} \end{pmatrix} \overset{*}{=} \perp \\
b = D^b(\widetilde{b}) &\triangleq \begin{pmatrix} \frac{(\alpha_l \cdot b + \beta_3) - \beta_3}{\alpha_l} \\ \frac{(\alpha_r \cdot b + \beta_4) - \beta_4}{\alpha_r} \end{pmatrix} = b \\
b' = D^b(\widetilde{a}') &\triangleq \begin{pmatrix} \frac{(\alpha_l \cdot a + \beta_1) - \beta_3}{\alpha_l} \\ \frac{(\alpha_r \cdot a + \beta_2) - \beta_4}{\alpha_r} \end{pmatrix} \triangleq \begin{pmatrix} a + \frac{\beta_1 - \beta_3}{\alpha_l} \\ a + \frac{\beta_2 - \beta_4}{\alpha_r} \end{pmatrix} \overset{*}{=} \perp
\end{aligned}
$$

*) Except for a negligible probability.

As shown above, decoding a well–formed DRAV using the wrong decoding function leads to $\perp$.

□

Lemma 3. *Without knowing the encryption keys, a non–well–formed DRAV is indistinguishable from a well–formed DRAV.*

Proof. From Lemma 1 we learn, that a well–formed DRAV only consists of two values uniformly at random. From Lemma 2 we learn, that well–formedness is a function of the encryption keys used to eventually decrypt the DRAV. Since a party that is not in possession of the encryption keys cannot know the decryption function, well–formed and non–well–formed DRAVs are indistinguishable for a party not in possession of the keys. □

2.3.2 DRAV Addition

This thesis refers to DRAV addition as *direct DRAV addition* because it has the property to be performable without knowledge of a DRAV's encryption keys. Although addition can be performed without the knowledge of the encryption keys, it will not reveal additional information because the encryption keys automatically change (see also Lemma 6). Therefore, the result will only be well–formed iff the addition was intended by the party in possession of the keys (see Lemma 2).

Lemma 4. *Additions commissioned by a party in possession of the encryption keys of well–formed DRAVs lead to well–formed DRAV:* $\widetilde{x} = (\widetilde{x_l}, \widetilde{x_r})$ *and* $\widetilde{y} = (\widetilde{y_l}, \widetilde{y_r})$ *can be added component–wise to* $\widetilde{z} = (\widetilde{x_1} + \widetilde{y_1}, \widetilde{x_2} + \widetilde{y_2})$*. The encryption keys for* $\widetilde{z}$ *will be* (α_l, α_r) *and* $(\beta_1+\beta_3, \beta_2+\beta_4)$ *assuming* $\widetilde{x}$ *was encrypted with* (α_l, α_r) *and* (β_1, β_2) *and* y *was encrypted with* (α_l, α_r) *and* (β_3, β_4)*.*

Proof. From $(\widetilde{x} = (\alpha_l \cdot x + \beta_1, \alpha_r \cdot x + \beta_2),\ \widetilde{y} = (\alpha_l \cdot x + \beta_3, \alpha_r \cdot x + \beta_4))$ it is obvious that $\widetilde{z} = (\alpha_l \cdot (x + y) + (\beta_1 + \beta_3), \alpha_r \cdot (x + y) + (\beta_2 + \beta_4))$ and $\widetilde{z}$ is well–formed. □

Lemma 5. *The addition of two DRAV leads to a non–well–formed DRAV if at least one operand DRAV is non–well–formed. It is of particular interest the following property holds:* $\forall \widetilde{x} : \widetilde{x} + \bot = \bot + \widetilde{x} = \bot$*.*

Proof. Let $\widetilde{\chi}$ be a non–well–formed DRAV, so: $\widetilde{\chi} = (\alpha_l \cdot \chi + \beta_3 + \Delta_l, \alpha_r \cdot \chi + \beta_4 + \Delta_r)$. The component–wise addition $\widetilde{\nu}$ of $\widetilde{\chi}$ to any well–formed $\widetilde{x} = (\alpha_l \cdot x + \beta_1, \alpha_r \cdot x + \beta_2)$ is $\widetilde{\nu} = (\alpha_l \cdot (x + \chi) + (\beta_1 + \beta_3) + \Delta_l, \alpha_r \cdot (x + \chi) + (\beta_2 + \beta_4) + \Delta_r)$. Using the component–wise universal decoding functions $D_l(\widetilde{v})$ and $D_r(\widetilde{v})$ (see Section 2.3), the value of χ is $\chi = (D_l(\widetilde{\chi_l}), D_r(\widetilde{\chi_r})) = (x + \chi + \frac{\Delta_l}{\alpha_l}, x + \chi + \frac{\Delta_r}{\alpha_r})$. So: $\forall(\Delta_r, \Delta_l) \in \mathbb{K} \setminus \{(0,0)\} \wedge \frac{\Delta_l}{\alpha_l} \neq \frac{\Delta_r}{\alpha_r} : D(\widetilde{\chi}) = \bot$. Since the encryption keys are not known to an attacker, $\frac{\Delta_l}{\alpha_l} \neq \frac{\Delta_r}{\alpha_r}$ holds except for a negligible probability if $\Delta_r \neq 0 \vee \Delta_r \neq 0$ and that's the property for being forged (non–well–formed) which was the assumption. Trivially $\bot + \bot = \bot$ by a similar argument. □

Lemma 6. *An addition which is unintended by a party in possession of the encryption keys does not reveal additional information. In other words: A party that is not in possession of the keys but that knows three well–formed DRAVs* $\widetilde{a}$*,* $\widetilde{b}$ *and* $\widetilde{c}$ *and is commissioned to execute the addition* $\widetilde{y} = \widetilde{a} + \widetilde{b}$ *cannot generate a well–formed DRAV* $\widetilde{y}' = \widetilde{a} + \widetilde{c}$*.*

Proof. Since every DRAV has per definition different dynamic keys, the component–wise addition of two DRAVs leads to a new dynamic key that is characteristic for the addition intended by a party in possession of the encryption keys.

Assuming three well–formed DRAVs $\widetilde{a}$, $\widetilde{b}$ and $\widetilde{c}$ and the party in possession of the keys commissions any other party to execute $\widetilde{y} = \widetilde{a} + \widetilde{b}$ but the other party tries to illicitly execute $\widetilde{y}' = \widetilde{a} + \widetilde{c}$:

$$\widetilde{a} = \begin{pmatrix} \alpha_l \cdot a + \beta_1 \\ \alpha_r \cdot a + \beta_2 \end{pmatrix}$$
$$\widetilde{b} = \begin{pmatrix} \alpha_l \cdot b + \beta_3 \\ \alpha_r \cdot b + \beta_4 \end{pmatrix}$$
$$\widetilde{c} = \begin{pmatrix} \alpha_l \cdot c + \beta_5 \\ \alpha_r \cdot c + \beta_6 \end{pmatrix}$$
$$\widetilde{y} = \begin{pmatrix} \alpha_l \cdot a + \beta_1 + \alpha_l \cdot b + \beta_3 \\ \alpha_r \cdot a + \beta_2 + \alpha_r \cdot b + \beta_4 \end{pmatrix} = \begin{pmatrix} \alpha_l \cdot (a+b) + (\beta_1 + \beta_3) \\ \alpha_r \cdot (a+b) + (\beta_2 + \beta_4) \end{pmatrix}$$
$$\widetilde{y}' = \begin{pmatrix} \alpha_l \cdot a + \beta_1 + \alpha_l \cdot c + \beta_5 \\ \alpha_r \cdot a + \beta_2 + \alpha_r \cdot c + \beta_6 \end{pmatrix} = \begin{pmatrix} \alpha_l \cdot (a+c) + (\beta_1 + \beta_5) \\ \alpha_r \cdot (a+c) + (\beta_2 + \beta_6) \end{pmatrix}$$

The following equations show the successful decoding of $\widetilde{y}$ and the unsuccessful decoding of $\widetilde{y}'$. This demonstrates, that an unsuccessful decoding leads to uniform randomness.

$$D^y(\widetilde{y}) = D(y) \qquad \text{(using } y\text{'s encryption keys)}$$
$$y = D^y(\widetilde{y}) \stackrel{\wedge}{=} \begin{pmatrix} \frac{(\alpha_l \cdot (a+b) + (\beta_1+\beta_3)) - (\beta_1+\beta_3)}{\alpha_l} \\ \frac{(\alpha_r \cdot (a+b) + (\beta_2+\beta_4)) - (\beta_2+\beta_4)}{\alpha_r} \end{pmatrix} = a + b$$
$$y' = D^y(\widetilde{y}') \stackrel{\wedge}{=} \begin{pmatrix} \frac{(\alpha_l \cdot (a+b) + (\beta_1+\beta_5)) - (\beta_1+\beta_3)}{\alpha_l} \\ \frac{(\alpha_r \cdot (a+b) + (\beta_2+\beta_6)) - (\beta_2+\beta_4)}{\alpha_r} \end{pmatrix} \stackrel{\wedge}{=} \begin{pmatrix} (a+b) + \frac{\beta_5 - \beta_3}{\alpha_l} \\ (a+b) + \frac{\beta_6 - \beta_4}{\alpha_r} \end{pmatrix} \stackrel{*}{=} \perp$$

*) Except for a negligible probability.

□

2.3.3 Final DRAV Decoding

Finally decoding a DRAV means to extract the hidden value which is crucial to successfully complete an evaluation. Obviously, ordinary parties have no information about the encryption keys and are thus not able to make use of the universal decoding function $D(\widetilde{v})$. This section describes a system that enables the party in possession of the keys to allow other parties to decode a specific DRAV. Usually it only allows the final DRAV to be decoded, otherwise intermediate information would leak.

The overall costs thereof are only one addition and one OAFE evaluation by the party not in possession of the keys. Of course, the OAFE has to be created by the party that knows the encryption keys. The setup of this OAFE can be seen as the authorization to decode exactly one specific DRAV. The procedure will decode a DRAV to a final scalar

value although D is a partial function. The special value $\perp$ (to which D evaluates if the DRAV is non–well–formed) is mapped to uniform randomness. As Lemma 2 proves, a DRAV is also non–well–formed if the encryption keys differ from the keys of the DRAV whose decoding was intended. Therefore, no other DRAVs can be decoded illegitimately. In the following, the DRAV to be decoded is named $\widetilde{v} = (\widetilde{v_l}, \widetilde{v_r})$ and the OAFE permitting the decoding is called the *final OAFE*.

The second party's input to the final OAFE is $\widehat{v} = \widetilde{v_l} + \widetilde{v_r}$, the addition of both components of the DRAV to decode. The final OAFE was created by the first party as follows: Assuming $\widetilde{v}$ is encrypted with (α_l, α_r) and (β_1, β_2), the first party knows $\widehat{v}$ has to be encrypted using $(\alpha_l + \alpha_r)$ and $(\beta_1 + \beta_2)$. Given this knowledge the final OAFE setup is $\frac{1}{\alpha_l+\alpha_r} \cdot \widehat{v} - \frac{\beta_1+\beta_2}{\alpha_l+\alpha_r}$. Note that this is the reason why the sum of the static keys of a DRAV has to be invertible (cf. Section 2.3).

Again, an attacker must be caught when trying to forge a DRAV: $D(\widetilde{v}) = \perp$ if $\widetilde{v}$ has been forged, $\perp$ is mapped to uniform randomness in this process since finite fields have no $\perp$–value. If the attacker cheats somewhere in the process and forges one of the DRAV tuples $\widetilde{x} = (\widetilde{x_1}, \widetilde{x_2})$ to $\widetilde{x'} = (\widetilde{x_1} + \Delta_1, \widetilde{x_2} + \Delta_2)$, the DRAV $\widetilde{x'}$ becomes (except for a negligible probability) non–well–formed. As shown in the equations below, the attempt to finally decode a non–well–formed DRAV results in randomness (assuming $\widetilde{x}$ is forged to $\widetilde{x'_1} = \widetilde{x_1} + \Delta_1$ and $\widetilde{x'_2} = \widetilde{x_2} + \Delta_2$):

$$
\begin{aligned}
\widehat{x'} =& \widetilde{x'_1} + \widetilde{x'_2} = \widetilde{x_1} + \Delta_1 + \widetilde{x_2} + \Delta_2 \\
\Rightarrow x' =& \frac{1}{\alpha_l + \alpha_r} \cdot \widehat{x'} - \frac{\beta_{x_1} + \beta_{x_2}}{\alpha_l + \alpha_r} \\
\Leftrightarrow x' =& \frac{\widetilde{x_1} + \Delta_1 + \widetilde{x_2} + \Delta_2}{\alpha_l + \alpha_r} - \frac{\beta_{x_1} + \beta_{x_2}}{\alpha_l + \alpha_r} \\
\Leftrightarrow x' =& \frac{(\alpha_l x + \beta_{x_1}) + \Delta_1 + (\alpha_r x + \beta_{x_2}) + \Delta_2}{\alpha_l + \alpha_r} - \frac{\beta_{x_1} + \beta_{x_2}}{\alpha_l + \alpha_r} \\
\Leftrightarrow x' =& \frac{(\alpha_l + \alpha_r)x + (\beta_{x_1} + \beta_{x_2} + \Delta_1 + \Delta_2)}{\alpha_l + \alpha_r} - \frac{\beta_{x_1} + \beta_{x_2}}{\alpha_l + \alpha_r} \\
\Leftrightarrow x' =& x + \frac{\beta_{x_1} + \beta_{x_2}}{\alpha_l + \alpha_r} + \frac{\Delta_1 + \Delta_2}{\alpha_l + \alpha_r} - \frac{\beta_{x_1} + \beta_{x_2}}{\alpha_l + \alpha_r} \\
\Leftrightarrow x' =& x + \frac{\Delta_1 + \Delta_2}{\alpha_l + \alpha_r}
\end{aligned}
$$

Because an attacker is not in possession of the static encryption keys α_l and α_r, it cannot control the value of $\frac{\Delta_1+\Delta_2}{\alpha_l+\alpha_r}$. Since α_l and α_r are chosen at random, the result x' becomes randomness, too. Note that α_l and α_r are not perfectly chosen at uniform randomness because α_l, α_r, and their sum have to be invertible (cf. Section 2.3). But, for large finite fields (such as $\mathbb{K}$) the distribution can still be considered uniformly at random.

Additionally, an attacker could choose $\Delta_l + \Delta_r = 0$ but because the value that gets passed to the OAFE is the addition of both values, $\Delta_l + \Delta_r = 0$ is honest behavior.

2.4 Dual Randomized Affine Encoding

Dual Randomized Affine Encodings (DRAEs) are affine representations of parts of arithmetic circuits. A DRAE describes arithmetic expressions as affine functions without revealing the original expression. A single DRAE alone is not powerful enough to describe arbitrary complex arithmetic expressions, therefore Section 2.5 proposes a procedure to evaluate arbitrary expression using multiple DRAEs. Technically, a DRAE is a specification of calculations that need to be performed to eventually obtain the evaluated value of the arithmetic expression. The reason for using DRAEs instead of evaluating the arithmetic expressions as usual is that a DRAE operates on encrypted values. Both, the input values and the output value of a DRAE are DRAVs (see Section 2.3). Additionally, the arithmetic expression performed is hidden from every party that is not in possession of the encryption keys. The reason DRAEs specify the calculations using affine functions is that OAFEs provide a secure way to obliviously evaluate affine functions. The procedure of evaluating arithmetic expression using DRAEs works as follows: The first party (which is in possession of the encryption keys) chooses the expression to evaluate. Next, it sets up the DRAE but hides the affine calculations inside OAFEs. In fact, instead of the affine calculations themselves, the DRAEs specifies references to the OAFEs that were configured to evaluate these functions obliviously. The evaluating party finally evaluates the DRAEs with the help of the OAFEs provided by the first party.

To easily envision the role of DRAEs and DRAVs: By performing arithmetic operations on DRAVs a DRAE evaluates an arithmetic expression to a resulting DRAV. Just the same way, an arithmetic expression performs operations on values and evaluates to a result value (e.g., given $x = 17$ and $y = 23$, the arithmetic expression $x * y + 3$ evaluates to 394).

Summing up, DRAE evaluation is partitioning calculations between the two parties: The first party which builds the DRAE and configures the OAFEs and the second which provides the input and does the evaluation. In fact, the first party is not directly involved in the evaluation process, it provided its calculations a priori hidden inside OAFEs. But as stated above, one DRAE cannot realize arbitrary complex arithmetic expressions, as a rule of thumb: As soon as a (non–trivial) multiplication is part of the original arithmetic, one DRAE is not enough to evaluate it.

The next sections describe in–depth how arithmetic operations can be safely encoded and how a DRAE describes this encoding. Since DRAEs evaluate to DRAVs and DRAVs can be easily added (see Section 2.3.2), DRAE addition is easy, too: A DRAE that encodes the addition of two DRAEs can be built practically for free. But DRAE multiplication is more complex: To multiply two DRAEs, both DRAEs have to be evaluated separately, their results have to be assigned to intermediate variables and a third DRAE has to be built that solely multiplies the intermediate variables. In contrast to general DRAE multiplication, a DRAE can encode the multiplication of two DRAVs and assigning intermediate values to variables is just giving intermediate DRAVs a temporary name. The multiplication encoding is an important contribution of this thesis and will be discussed in–depth in Section 2.4.2.

2.4.1 Encoding Additions

Since two DRAVs can directly be added as explained in section 2.3.2 and DRAEs evaluate to DRAVs, two DRAEs can be added directly as well. Notably, although the encryption keys change, this operation can be performed by any party, in possession of the encryption keys or not. Similarly to DRAVs, unintended operations ruin the information because the encryption keys change when performing operations. Operations unintended by the party in possession of the keys lead to non–well–formed DRAVs because wrong encryption keys will be used to decode them.

2.4.2 Encoding Multiplications

Since neither a DRAV (Section 2.3) nor a DRAE can be multiplied directly, most multiplications have to be performed using a construction involving additional DRAEs and intermediate variables holding DRAVs. Section 2.5 describes the transformation of arbitrary arithmetic expressions. This section describes multiplications that can directly be expressed as a DRAE: A single multiplication of two DRAVs $\widetilde{x}$ and $\widetilde{y}$ each assigned to an intermediate variable. The process that lead to the final multiplication encoding is described in the next section, followed by a description of the final encoding.

2.4.2.1 Towards Secure Multiplications

This section describes the iterations passed through until reaching the goal of secure multiplications. The description starts from ground up only using basic finite field mathematics.

The general construction is inspired by the paper *How to Garble Arithmetic Circuits* by Applebaum et al. [AIK12]. In fact, the first version which appears in the following equations is the same as Applebaum's version. Applebaum's version enables to encode multiplications without revealing the operands. The further iterations which are presented thereafter are derivations specially developed for this thesis.

For input values x and y, the following OAFE configuration can be used to perform the multiplication $x \cdot y$.

$$\begin{pmatrix}\mathfrak{A}\\ \mathfrak{C}\end{pmatrix} = \begin{pmatrix}1\\ r_2\end{pmatrix} \cdot x + \begin{pmatrix}-r_1\\ -r_1 r_2 + r_3\end{pmatrix} \qquad \begin{pmatrix}\mathfrak{B}\\ \mathfrak{D}\end{pmatrix} = \begin{pmatrix}1\\ r_1\end{pmatrix} \cdot y + \begin{pmatrix}-r_2\\ -r_3\end{pmatrix}$$

The evaluating party is then enabled to calculate the result $z = x \cdot y$ without revealing information about x or y.

$$z = \mathfrak{A} \cdot \mathfrak{B} + \mathfrak{C} + \mathfrak{D} = x \cdot y$$

For uniformly random numbers $r_1, r_2, r_3, r_4 \in \mathbb{K}$ the terms $\mathfrak{A}$, $\mathfrak{B}$, $\mathfrak{C}$, and $\mathfrak{D}$ can be calculated using two OAFEs and every term alone does not reveal any information because

each is one–time encrypted. But their composition ($\mathfrak{A}\cdot\mathfrak{B}+\mathfrak{C}+\mathfrak{D}$) evaluates to the desired result $x \cdot y$. Because of the one–time encryption, a simple two party evaluation could be realized easily by calculating $\mathfrak{A}$ and $\mathfrak{C}$ or $\mathfrak{B}$ and $\mathfrak{D}$ a priori and delivering the values to the other party. Using one OAFE configured by the first party, the second party is then able to calculate the other terms and therefore the result $x \cdot y$. Exemplary, the first party draws three random numbers, say $r_1 = 7, r_2 = 5, r_3 = 3$ and chooses its input $x = 23$. Next, it calculates the values for $\mathfrak{A} = x - r_1 = 23 - 7 = 16$, $\mathfrak{C} = 83$, and configures one OAFE that calculates $\mathfrak{B}$ and $\mathfrak{D}$. The final step is the transmission of $\mathfrak{A} = 16$, $\mathfrak{C} = 83$ and the OAFE to the second party. The first party has now completed its acting. The second party starts by evaluating the received OAFE with its input, say $y = 42$. The OAFE would obliviously evaluate to $\mathfrak{B} = 37$ and $\mathfrak{D} = 291$. Now, the second party knows everything it needs to compute $x \cdot y = 23 \cdot 42 = 966 = 16 \cdot 37 + 83 + 291 = \mathfrak{A} \cdot \mathfrak{B} + \mathfrak{C} + \mathfrak{D}$. So, the second party is able to calculate $x \cdot y$ without knowing x. Unfortunately, this construction is not feasible for composable computations because the second party learns the result of every multiplication: The multiplication result is potentially an intermediate value the second party should not learn. This flaw can be removed by a simple refinement that will be discussed in the next section.

Encrypted Variables are introduced to hide intermediate values. The basic idea is to have "boxed" variables $\bar{x}$ which can only be "looked into" with the keys. Syntactically $\bar{x}$ stands for the encrypted (boxed) form of x. x is encrypted using two random values α ($\neq 0$) and β to $\bar{x} = \alpha x + \beta$. For the sake of simplicity, the "boxing" function $B(x) = \alpha x + \beta = \bar{x}$ and the "unboxing" function $U(\bar{x}) = \frac{\bar{x}-\beta}{\alpha}$ are used hereafter. The function B is assumed to generate appropriate keys automatically and U is assumed to implicitly know the encryption keys from the context. Using the encrypted values, the former multiplication encoding can be revised to also keep secret the evaluated result. To be able to use this multiplication encoding, the input variables x and y have to be encrypted to $\bar{x}$ and $\bar{y}$ first. This is achieved by providing additional OAFEs to lift the plain input values into the "encrypted world".

$$\bar{x} = B(x) = \alpha_x x + \beta_x \qquad\qquad \bar{y} = B(x) = \alpha_x y + \beta_y$$

As before, the calculation itself is also performed via OAFEs.

$$\begin{pmatrix}\mathfrak{A}\\ \mathfrak{C}\end{pmatrix} = \begin{pmatrix}\alpha_z\\ \alpha_z r_2\end{pmatrix} \cdot U(\bar{x}) + \begin{pmatrix}-r_1\\ -r_1 r_2 + r_3 + \beta_z\end{pmatrix} \qquad \begin{pmatrix}\mathfrak{B}\\ \mathfrak{D}\end{pmatrix} = \begin{pmatrix}1\\ r_1\end{pmatrix} \cdot U(\bar{y}) + \begin{pmatrix}-r_2\\ -r_3\end{pmatrix}$$

Because the composition of affine functions leads to affine functions as well, the calculation above can be evaluated using two OAFEs ($\bar{x}$ being the input for the first OAFE, $\bar{y}$ the input to the second OAFE). The final step after having evaluated the OAFEs is the calculation of the encrypted value $\bar{z} = \bar{x} \cdot \bar{y}$.

$$\bar{z} = \mathfrak{A} \cdot \mathfrak{B} + \mathfrak{C} + \mathfrak{D} (= B(U(x) \cdot U(y)) = \alpha_z xy + \beta_z)$$

The complete calculation of $\bar{z} = B(x) \cdot B(y)$ needs four OAFEs: two to initially encrypt the inputs $\bar{x}$ and $\bar{y}$ and two for the multiplication $\bar{x} \cdot \bar{y}$. In the same way as before, less OAFEs are needed if one or more inputs come from the party that configures the OAFEs. In the following examination of two–party computations, the first party (which generates the OAFEs) is named *Goliath* and the other party (which does the evaluation) is named *David*. Of course, David has no knowledge of the encryption keys, the random numbers, and the OAFE configuration. At the beginning, Goliath draws the random numbers $\alpha_x, \alpha_y, \alpha_z \in \mathbb{K} \setminus \{0\}$ and $\beta_x, \beta_y, \beta_z, r_1, r_2, r_3 \in \mathbb{K}$ uniformly at random. Goliath then configures the OAFEs appropriately and after the multiplication, $\bar{z}$ is encrypted using the encryption keys α_z and β_z. Finally, Goliath transmits the OAFEs and the values of the terms that only depend on its own inputs. Next, David uses the OAFEs appropriately and calculates $\bar{z}$. The benefit of this construction is composability: By instructing David to store (encrypted) intermediate values which David will provide in further calculations, arbitrary multiplicative expressions can be modeled with this construction. Exemplary, the evaluation of the expression $(x \cdot y) \cdot z$ is discussed. Assuming David provides x and Goliath provides y and z, the expression can be evaluated by first boxing David's input $\bar{x} = B(x)$ using one OAFE. Using the described construction and given that it obtained $\bar{y}$ and $\bar{z}$, David can now calculate the encrypted intermediate result $\bar{t} = \bar{x} \cdot \bar{y}$ without knowing anything about y or the plain intermediate result value t. In the next step, David is commissioned to calculate $\bar{o} = \bar{t} \cdot \bar{z}$ similarly, $\bar{o}$ being the final result. Using one additional OAFE, Goliath can allow David to execute $o = U(\bar{o})$ yielding the unencrypted result $o = (x \cdot y) \cdot z$. David will not learn anything about y, t or z that it cannot calculate from the output o and his input x anyways. David could try to forge the value of $\bar{t}$ before calculating $\bar{t} \cdot \bar{z}$ but because it has no information about the encryption keys of $\bar{t}$ or $\bar{z}$ it will render $o = U(\bar{o})$ to randomness because it will contain random terms depending on some encryption key $(\forall \bar{v} \in \{B(v) \mid v \in \mathbb{K}\}, \Delta \in \mathbb{K} \setminus \{0\} : U(\bar{v} + \Delta) \neq U(\bar{v}) + \Delta)$. However, there is a security flaw: Usually, from $o = (x \cdot y) \cdot z = 0$, David can learn $x = 0 \vee y = 0 \vee z = 0$. Because x is its own input, when knowing $x \neq 0$, from $o = 0$ it should learn $y = 0 \vee z = 0$. Unfortunately, David could choose $\bar{t}$ randomly and with overwhelming probability $U(\bar{t}) \neq 0$. So forging $\bar{t}$ to a random number will reveal additional information to David: $o = 0 \Rightarrow U(\bar{t} \cdot \bar{z}) = 0 \Rightarrow z = 0$ with overwhelming probability. This means that even though David is not aware of the concrete change it does to the calculation, it can reveal additional information when the result of a multiplication is 0. In other words, David may be able to test whether intermediate results are zero. Because this flaw violates the requirement that every party should only learn the result, the multiplication encoding underwent the next iteration.

Using Message Authentication Codes illicit changes to intermediate values can be prevented. The original proposal of Carter and Wegman [CW79] to hash integers is to choose a prime p (and random keys $a \neq 0$ and b) and use

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

In finite fields (such as $\mathbb{K}$) the modulo operations can be removed and the hashing function can be adapted to $h_{a,b}(x) = (ax + b)$ being an universal hashing function. For as and bs only used once, the hashing function is a one–time encryption as well. Because this thesis

uses $\bar{x} = B(x) = \alpha x + \beta$ with fresh α and β for every variable, $\bar{x}$ is an one–time encryption of x. Because of the flaw which reveals additional information when multiplications result in 0, this iteration revises boxed values and switches to DRAVs (Section 2.3).

DRAVs are twofold one–time encryptions that authenticate each other pairwise. This means that a DRAV is only valid (well–formed) iff the decryption of both one–time encryptions are equal. Hence, DRAVs are encrypted values with *Message Authentication Codes* (MACs) build–in. Syntactically, $\widetilde{x} = (\widetilde{x_l}, \widetilde{x_r})$ is the DRAV representing the plain value x. $\widetilde{x_l}$ and $\widetilde{x_r}$ are the one–time encryptions and the pairwise MACs. To be able to multiply two DRAVs $\widetilde{x}$ and $\widetilde{y}$, the multiplication construction is adapted. Before performing calculations involving DRAVs, the plain values have to be initially encrypted as DRAVs using OAFEs:

$$\widetilde{x} = E(x) = \begin{pmatrix} \alpha_l \\ \alpha_r \end{pmatrix} \cdot x + \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} \qquad \widetilde{y} = E(y) = \begin{pmatrix} \alpha_l \\ \alpha_r \end{pmatrix} \cdot y + \begin{pmatrix} \beta_3 \\ \beta_4 \end{pmatrix}$$

OAFEs configured as following can then be used to perform the multiplication $\widetilde{x} \cdot \widetilde{y}$.

$$\begin{pmatrix} \mathfrak{A} \\ \mathfrak{G} \end{pmatrix} = \begin{pmatrix} \alpha_l \\ \alpha_r r_6 \end{pmatrix} \cdot D_l(\widetilde{x}) + \begin{pmatrix} -r_1 \\ -r_7 \end{pmatrix} \qquad \begin{pmatrix} \mathfrak{B} \\ \mathfrak{H} \end{pmatrix} = \begin{pmatrix} 1 \\ r_5 \end{pmatrix} \cdot D_l(\widetilde{y}) + \begin{pmatrix} -r_2 \\ r_8 \end{pmatrix}$$
$$\begin{pmatrix} \mathfrak{E} \\ \mathfrak{C} \end{pmatrix} = \begin{pmatrix} \alpha_r \\ \alpha_l r_2 \end{pmatrix} \cdot D_r(\widetilde{x}) + \begin{pmatrix} -r_5 \\ r_3 \end{pmatrix} \qquad \begin{pmatrix} \mathfrak{F} \\ \mathfrak{D} \end{pmatrix} = \begin{pmatrix} 1 \\ r_1 \end{pmatrix} \cdot D_r(\widetilde{y}) + \begin{pmatrix} -r_6 \\ r_4 \end{pmatrix}$$

The encryption keys β_5 and β_6 arise from the random numbers drawn before configuring the OAFEs.

$$\beta_5 = r_1 r_2 + r_3 + r_4 \qquad \beta_6 = r_5 r_6 + r_7 + r_8$$

The evaluating party is finally able to calculate the DRAV $\widetilde{z} = \widetilde{x} \cdot \widetilde{y}$ only from the OAFE outputs $\mathfrak{A}$ to $\mathfrak{H}$.

$$\widetilde{z} = \begin{pmatrix} \mathfrak{A} \cdot \mathfrak{B} + \mathfrak{C} + \mathfrak{D} \\ \mathfrak{E} \cdot \mathfrak{F} + \mathfrak{G} + \mathfrak{H} \end{pmatrix} = \begin{pmatrix} \alpha_l xy + \beta_5 \\ \alpha_r xy + \beta_6 \end{pmatrix}$$

Note that α_l and α_r have the same value in every DRAV. This is a desired property of DRAVs: They are encrypted using constant static keys α_l and α_r and dynamic keys β and β' that are only shared with other DRAVs by chance. For details see Section 2.3.

As before, all terms can be evaluated using OAFEs and the construction is composable by commissioning David to store intermediate DRAVs. The following equations examine the multiplication of modified DRAVs. Modifying DRAVs means to illicitly change their encrypted values. The potential changes manifest in the Δ values.

$$
\begin{aligned}
\widetilde{z} &= \begin{pmatrix} (\alpha_l D_l(\widetilde{x}) - r_1)(D_l(\widetilde{y}) - r_2) \\ (\alpha_r D_r(\widetilde{x}) - r_5)(D_r(\widetilde{y}) - r_6) \end{pmatrix} + \begin{pmatrix} \alpha_l r_2 D_r(\widetilde{x}) + r_3 + r_1 D_r(\widetilde{y}) + r_4 \\ \alpha_r r_6 D_l(\widetilde{x}) + r_7 + r_5 D_l(\widetilde{y}) + r_8 \end{pmatrix} \\
&= \begin{pmatrix} \alpha_l D_l(\widetilde{x}) D_l(\widetilde{y}) - \alpha_l r_2 D_l(\widetilde{x}) - r_1 D_l(\widetilde{y}) + r_1 r_2 \\ \alpha_r D_r(\widetilde{x}) D_r(\widetilde{y}) - \alpha_r r_6 D_r(\widetilde{x}) - r_5 D_r(\widetilde{y}) + r_5 r_6 \end{pmatrix} \\
&\quad + \begin{pmatrix} \alpha_l r_2 D_r(\widetilde{x}) + r_3 + r_1 D_r(\widetilde{y}) + r_4 \\ \alpha_r r_6 D_l(\widetilde{x}) + r_7 + r_5 D_l(\widetilde{y}) + r_8 \end{pmatrix} \\
&= \begin{pmatrix} \alpha_l \left(x + \frac{\Delta_1}{\alpha_l}\right)\left(y + \frac{\Delta_3}{\alpha_l}\right) - \alpha_l r_2 \left(x + \frac{\Delta_1}{\alpha_l}\right) - r_1 \left(y + \frac{\Delta_3}{\alpha_l}\right) + r_1 r_2 \\ \alpha_r \left(x + \frac{\Delta_2}{\alpha_r}\right)\left(y + \frac{\Delta_4}{\alpha_r}\right) - \alpha_r r_6 \left(x + \frac{\Delta_2}{\alpha_r}\right) - r_5 \left(y + \frac{\Delta_4}{\alpha_r}\right) + r_5 r_6 \end{pmatrix} \\
&\quad + \begin{pmatrix} \alpha_l r_2 \left(x + \frac{\Delta_2}{\alpha_r}\right) + r_3 + r_1 \left(y + \frac{\Delta_4}{\alpha_r}\right) + r_4 \\ \alpha_r r_6 \left(x + \frac{\Delta_1}{\alpha_l}\right) + r_7 + r_5 \left(y + \frac{\Delta_3}{\alpha_l}\right) + r_8 \end{pmatrix} \\
&= \begin{pmatrix} \alpha_l(xy) + (r_1 r_2 + r_3 + r_4) + \Delta_3 x + \Delta_1 y + \frac{\Delta_1 \Delta_3}{\alpha_l} - \Delta_1 r_2 + \frac{\Delta_2 \alpha_l r_2}{\alpha_r} - \frac{\Delta_3 r_1}{\alpha_l} + \frac{\Delta_4 r_1}{\alpha_r} \\ \alpha_r(xy) + (r_5 r_6 + r_7 + r_8) + \Delta_4 x + \Delta_2 y + \frac{\Delta_2 \Delta_4}{\alpha_r} + \frac{\Delta_1 \alpha_l r_6}{\alpha_l} - \Delta_2 r_6 + \frac{\Delta_3 r_5}{\alpha_l} - \frac{\Delta_4 r_5}{\alpha_r} \end{pmatrix}
\end{aligned}
$$

This leads to two possibilities which will be discussed in detail: Either the inputs $\widetilde{x}$ and $\widetilde{y}$ are both well–formed or not.

1. The DRAVs $\widetilde{x}$ and $\widetilde{y}$ are well–formed. This leads to a DRAV encoding the desired result $E(x \cdot y)$. The resulting encryption keys for $\widetilde{z}$ are α_l, α_r (static keys) and $\beta = r_1 r_2 + r_3 + r_4$, $\beta' = r_5 r_6 + r_7 + r_8$ (dynamic keys).

   $$\Delta_1 = \Delta_2 = \Delta_3 = \Delta_4 = 0 \Rightarrow \widetilde{z} = \begin{pmatrix} \alpha_l(xy) + (r_1 r_2 + r_3 + r_4) \\ \alpha_r(xy) + (r_5 r_6 + r_7 + r_8) \end{pmatrix}$$

   The decrypted value is $z = D(\widetilde{z}) = xy$ as expected.

2. At least one of the DRAVs $\widetilde{x}$ and $\widetilde{y}$ is non–well–formed. If at least one DRAV is non–well–formed, at least one Δ is non–zero. This leads to a non–well–formed DRAV $\widetilde{z}$ whose both components are uniformly at random.

   $$
   \begin{aligned}
   &\Delta_1 \neq 0 \vee \Delta_2 \neq 0 \vee \Delta_3 \neq 0 \vee \Delta_4 \neq 0 \Rightarrow \\
   &\widetilde{z} = \begin{pmatrix} \alpha_l(xy) + (r_1 r_2 + r_3 + r_4) + \Delta_3 x + \Delta_1 y + \frac{\Delta_1 \Delta_3}{\alpha_l} - \Delta_1 r_2 + \frac{\Delta_2 \alpha_l r_2}{\alpha_r} - \frac{\Delta_3 r_1}{\alpha_l} + \frac{\Delta_4 r_1}{\alpha_r} \\ \alpha_r(xy) + (r_5 r_6 + r_7 + r_8) + \Delta_4 x + \Delta_2 y + \frac{\Delta_2 \Delta_4}{\alpha_l} + \frac{\Delta_1 \alpha_l r_6}{\alpha_r} - \Delta_2 r_6 + \frac{\Delta_3 r_5}{\alpha_r} - \frac{\Delta_4 r_5}{\alpha_l} \end{pmatrix}
   \end{aligned}
   $$

   As one can easily see, if at least one $\Delta \neq 0$ the decoded result ($z = D(\widetilde{z})$) will comprise at least one term that depends on fresh, uniform randomness ($r_{1\ldots8}$). In a finite field this means that both components are uniformly at random. Additionally the problem with multiplications that evaluate to 0 is fixed in this version.

However, this process still has one potential security problem: The resulting term is potentially not uncorrelated to the OAFE outputs ($\mathfrak{A}$ to $\mathfrak{H}$) even when the attacker forged a DRAV. Assuming, the attacker forged at least $\widetilde{x_l}$. This means $\Delta_1 \neq 0$ and from $\mathfrak{A}$ an attacker learns:

$$\mathfrak{A} = \alpha_l \left(x + \frac{\Delta_1}{\alpha_l} \right) - r_1 \Leftrightarrow r_1 = \alpha_l x + \Delta_1 - \mathfrak{A}$$

Because an attacker knows Δ_1, $\mathfrak{A}$ (OAFE output) and the environment (see Section 3 for details) may know x, it can reveal information about the secret random number r_1. The problem is that r_1 is needed to turn the overall error terms to uniform and uncorrelated randomness which is potentially not true if too much information leaks about r_1. Worse, similar arguments hold for the other OAFE outputs ($\mathfrak{B}$ to $\mathfrak{H}$) and the secret random numbers $r_{2\ldots8}$. Therefore, the multiplication encoding is improved again to its final version which is described in Section 2.4.2.2.

2.4.2.2 Secure Multiplication Construction

The security flaw of the last vulnerable multiplication construction (see last construction in Section 2.4.2.1) is a potential correlation between information obtainable from OAFE outputs and the errors terms if David forges DRAVs. Hence, this final multiplication construction adds uncorrelated random values to every OAFE output if the input was forged. The general idea is to split the OAFE output in two parts called *radicals*. The sum of the two radicals that belong together has the same value as the OAFE outputs in the last vulnerable construction. Obviously, the value is only the same iff David did not try to forge the DRAVs.

The radicals trick affects OAFE calculations such as $\Phi \cdot D_\delta(\widetilde{v}) + \Psi$ ($\delta \in \{l, r\}$ being the tuple component (left/right), $\widetilde{v}$ a DRAV, and $\Phi, \Psi \in \mathbb{K}$ the OAFE parameters except decoding). Each calculation is divided into two calculations—the radicals—named R_1 and R_2 in the equations below. The values κ and γ are fresh values uniformly at random, which are only used to protect one OAFE calculation $\Phi \cdot D_\delta(\widetilde{v}) + \Psi$. The meaning of $!\delta$ is "the other tuple component": $!\delta = l \Leftrightarrow \delta = r \wedge !\delta = r \Leftrightarrow \delta = l$.

$$\begin{aligned} R_1 &= (1 - \kappa) \cdot (\Phi \cdot D_\delta(\widetilde{v}) + \Psi) + \gamma \\ R_2 &= (\kappa) \cdot (\Phi \cdot D_{!\delta}(\widetilde{v}) + \Psi) - \gamma \end{aligned}$$

The benefit of R_1 and R_2 is their sum: $R_1 + R_2 = \Phi \cdot D_\delta(\widetilde{v}) + \Psi$ if and only if $D_\delta(\widetilde{v}) = D_{!\delta}(\widetilde{v})$. This is the condition whether $\widetilde{v}$ is well–formed or not (see Section 2.3.1). If $\widetilde{v}$ is non–well–formed , $R_1 + R_2$ will contain terms depending on κ.

The combination of the radicals trick and the former version is given in the equations below. This is the last iteration in the development of secure multiplication for this thesis.

The first step is, just the same way as in the insecure constructions before, to encode the input values x and y as DRAVs $\widetilde{x}$ and $\widetilde{y}$.

$$\widetilde{x} = E(x) = (\alpha_l x + \beta_1, \alpha_r x + \beta_2)$$
$$\widetilde{y} = E(x) = (\alpha_l y + \beta_3, \alpha_r y + \beta_4)$$

The DRAV multiplication is, as before, done by evaluating OAFEs which were configured for this purpose ($\odot$ denotes component–wise multiplication).

$$\begin{pmatrix}\mathfrak{A}_1\\ \mathfrak{G}_1\\ \mathfrak{E}_2\\ \mathfrak{C}_2\end{pmatrix} = \begin{pmatrix}1-\kappa_1\\ 1-\kappa_2\\ \kappa_5\\ \kappa_6\end{pmatrix} \odot \left(\begin{pmatrix}\alpha_l\\ \alpha_r r_6\\ \alpha_r\\ \alpha_l r_2\end{pmatrix} \cdot D_l(\widetilde{x}) + \begin{pmatrix}-r_1\\ -r_7\\ -r_5\\ r_3\end{pmatrix} \right) + \begin{pmatrix}\gamma_1\\ \gamma_2\\ -\gamma_5\\ -\gamma_6\end{pmatrix}$$
$$\begin{pmatrix}\mathfrak{B}_1\\ \mathfrak{H}_1\\ \mathfrak{F}_2\\ \mathfrak{D}_2\end{pmatrix} = \begin{pmatrix}1-\kappa_3\\ 1-\kappa_4\\ \kappa_7\\ \kappa_8\end{pmatrix} \odot \left(\begin{pmatrix}1\\ r_5\\ 1\\ r_1\end{pmatrix} \cdot D_l(\widetilde{y}) + \begin{pmatrix}-r_2\\ r_8\\ -r_6\\ r_4\end{pmatrix} \right) + \begin{pmatrix}\gamma_3\\ \gamma_4\\ -\gamma_7\\ -\gamma_8\end{pmatrix}$$
$$\begin{pmatrix}\mathfrak{E}_1\\ \mathfrak{C}_1\\ \mathfrak{A}_2\\ \mathfrak{G}_2\end{pmatrix} = \begin{pmatrix}1-\kappa_5\\ 1-\kappa_6\\ \kappa_1\\ \kappa_2\end{pmatrix} \odot \left(\begin{pmatrix}\alpha_r\\ \alpha_l r_2\\ \alpha_l\\ \alpha_r r_6\end{pmatrix} \cdot D_r(\widetilde{x}) + \begin{pmatrix}-r_5\\ r_3\\ -r_1\\ -r_7\end{pmatrix} \right) + \begin{pmatrix}\gamma_5\\ \gamma_6\\ -\gamma_1\\ -\gamma_2\end{pmatrix}$$
$$\begin{pmatrix}\mathfrak{F}_1\\ \mathfrak{D}_1\\ \mathfrak{B}_2\\ \mathfrak{H}_2\end{pmatrix} = \begin{pmatrix}1-\kappa_7\\ 1-\kappa_8\\ \kappa_3\\ \kappa_4\end{pmatrix} \odot \left(\begin{pmatrix}1\\ r_1\\ 1\\ r_5\end{pmatrix} \cdot D_r(\widetilde{y}) + \begin{pmatrix}-r_6\\ r_4\\ -r_2\\ r_8\end{pmatrix} \right) + \begin{pmatrix}\gamma_7\\ \gamma_8\\ -\gamma_3\\ -\gamma_4\end{pmatrix}$$

As before, the calculations can still be performed using (four) OAFEs and the encryption keys β_5 and β_6 for the resulting DRAV $\widetilde{z}$ yield as $\beta_5 = r_1 r_2 + r_3 + r_4$ and $\beta_6 = r_5 r_6 + r_7 + r_8$. The reunification of the radicals obtained from the OAFE evaluations yield the result $\widetilde{z}$.

$$\widetilde{z} = \begin{pmatrix}(\mathfrak{A}_1 + \mathfrak{A}_2)\cdot(\mathfrak{B}_1 + \mathfrak{B}_2) + (\mathfrak{C}_1 + \mathfrak{C}_2) + (\mathfrak{D}_1 + \mathfrak{D}_2)\\ (\mathfrak{E}_1 + \mathfrak{E}_2)\cdot(\mathfrak{F}_1 + \mathfrak{F}_2) + (\mathfrak{G}_1 + \mathfrak{G}_2) + (\mathfrak{H}_1 + \mathfrak{H}_2)\end{pmatrix} = \begin{pmatrix}\alpha_l xy + \beta_5\\ \alpha_r xy + \beta_6\end{pmatrix}$$

To analyze the security of this multiplication encoding, the following equations exemplarily analyze the term $(\mathfrak{A}_1 + \mathfrak{A}_2)$. Assuming the attacker either changes the value of the left tuple component ($\Rightarrow \Delta_1 \neq 0$) or the right component ($\Rightarrow \Delta_2 \neq 0$), $\widetilde{x}$ is non–well–formed.

$$
\begin{aligned}
\mathfrak{A}_1 &= (1-\kappa_1)\cdot(\alpha_l \cdot D_l(\widetilde{x}) - r_1) + \gamma_1 \\
&= (1-\kappa_1)\cdot\left(\alpha_l \cdot \left(x + \frac{\Delta_1}{\alpha_l}\right) - r_1\right) + \gamma_1 \\
&= (1-\kappa_1)\cdot(\alpha_l x + \Delta_1 - r_1) + \gamma_1 \\
&= (1-\kappa_1)\cdot(\alpha_l x - r_1) + \gamma_1 + (\Delta_1 - \kappa_1\Delta_1) \\
\mathfrak{A}_2 &= \kappa_1 \cdot (\alpha_l \cdot D_l(\widetilde{x}) - r_1) + -\gamma_1 \\
&= \kappa_1 \cdot \left(\alpha_l \cdot \left(x + \frac{\Delta_2}{\alpha_r}\right) - r_1\right) + -\gamma_1 \\
&= \kappa_1 \cdot \left(\alpha_l x + \frac{\alpha_l \Delta_2}{\alpha_r} - r_1\right) + -\gamma_1 \\
&= \kappa_1 \cdot (\alpha_l x - r_1) + -\gamma_1 + \left(\frac{\kappa_1 \alpha_1 \Delta_2}{\alpha_r}\right) \\
\mathfrak{A}_1 + \mathfrak{A}_2 &= \alpha_l x - r_1 + \left(\Delta_1 - \kappa_1\Delta_1 + \frac{\kappa_1\alpha_1}{\alpha_r}\Delta_2\right) \\
&= \alpha_l x - r_1 + \left(\kappa_1 \cdot \left(\frac{\Delta_1}{\kappa_1} - \Delta_1 + \frac{\alpha_l}{\alpha_r}\Delta_2\right)\right)
\end{aligned}
$$

Because an attacker cannot know α_l, α_r, or κ_1, the error term will (except for the negligible probability) contain κ_1. Obviously, if $\Delta_1 = \Delta_2 = 0$ the attacker was honest and no error term gets generated. Because κ_1 is uncorrelated to any other term in the whole process, the DRAV will be made of uniform randomness, if an attacker tries to cheat.

2.5 Dual Randomized Affine Circuits

Dual Randomized Affine Circuits (DRACs) are representations of entire arbitrary arithmetic circuits or functions. As discussed in Section 2.4, not every arithmetic function can be expressed as one DRAE. Therefore, the overall function is divided in as many DRAEs as needed, all of which are assigned to intermediate variables. Every DRAE can use the evaluated intermediate variables of preceding DRAEs if the computation was too complex for a single DRAE. This resembles to evaluate the simple function $f(x) = (a \cdot b) + (c \cdot d)$ as a sequence of sub–expressions, each assigned to an intermediate variable: $t_0 \leftarrow a \cdot b; t_1 \leftarrow c \cdot d; y \leftarrow t_0 + t_1$. In just the same way, DRACs are sequences of DRAEs, each assigned to a variable. Because DRAEs evaluate to DRAVs, the variables will hold a DRAV as soon as the DRAE was evaluated. Obviously, the DRAC evaluation has to be performed in the correct order, otherwise some variables may not have their values assigned yet when needed. Note that a DRAC is not limited to encoding polynomials or arithmetic functions, a DRAC can have more than one output DARE and therefore model arbitrary arithmetic circuits. However, the security is only proved for Oblivious Polynomial Evaluation and the implementation expects arithmetic functions (arithmetic circuits with only one output node).

2.5.1 Evaluating a DRAC

A DRAC evaluates to its value by evaluating one DRAE after the other and assigning the resulting DRAV to the variable the DRAE is assigned to. Before the evaluation of the first DRAE, all (unencrypted) function parameters are evaluated to DRAVs using the provided OAFEs. The DRAE sequence is ordered, so a DRAE only needs values (DRAVs) of DRAEs evaluated before. The last DRAV can then be decoded as explained in Section 2.3.3. The result from the decoding of the final DRAE is the result of the whole computation or uniform randomness (meaning $\perp$).

2.6 Oblivious Polynomial Evaluation

This section describes the *Oblivious Polynomial Evaluation* (OPE) of a univariate polynomial $f(x) = \sum_{i=0}^{k} a_i x^i$. OPE is the joint evaluation of a polynomial that is only known by the first party at a node that is only known by the second party. The evaluation should neither reveal the polynomial to the second party nor reveal the node to the first party.

Using techniques described in this thesis, the first party (Goliath) starts the process by choosing the polynomial coefficients and transforming f into a DRAC (see Section 2.5). The number of DRAEs the DRAC consists of is in $O(n)$, n being the number of arithmetic operations needed to evaluate f as usual. Using Horner's rule [CLRS01], evaluating a polynomial of degree n only requires n additions and n multiplications. Next, Goliath calculates the OAFE setup. Note that Goliath has to accumulate all OAFE calls using the same variable to one OAFE call resulting in a vector. Otherwise, a corrupted party could evaluate the OAFEs with different values for one and the same variable.

After the first party has calculated the DRAC and the OAFE configuration, it sends the DRAC along with the OAFE to the second party (David). Naturally, every affine function inside the DRAEs is replaced by a reference to an OAFE (and the position in the resulting vector) because otherwise the encryption keys would be revealed to the second party.

The DRAC is transmitted using a signed and encrypted channel. The identity of the party the DRAC is transferred to does not need to be verified because a DRAC does not contain any secrets. A DRAC is made of DRAEs that only refer to uncorrelated uniform randomness and OAFEs.

After having received the DRAC and the OAFE references, the second party can evaluate the DRAC. It evaluates the DRAEs one by one and assigns the resulting DRAVs to their intermediate variables as described in the DRAC. As explained in Sections 2.3.3 and 2.5.1 both components of the final DRAV have then to be added and passed to the final OAFE which yields the unencrypted evaluated value of the polynomial.

2.7 Verifying the Polynomial Degree

From a given DRAE it is easily possible to calculate the maximum polynomial degree of the function the DRAE encodes. Since multiplications are performed via OAFEs while evaluating the DRAE, the evaluating party is able to keep track of the maximum degree

of the overall function. The following steps describe the process on how to calculate the maximum degree of the function a DRAE encodes.

- Constant terms are annotated with degree 0.
- Terms that refer to the initial function parameter(s) are annotated with degree 1.
- The multiplicative DRAE terms are annotated with the sum of both (intermediate) variables they multiply.
- The additive DRAE terms are annotated with the degree of the (intermediate) variable they refer to.
- The last step is to annotate the DRAE itself with the maximum annotation of all its descendants.

Since DRACs that evaluate polynomials always have exactly one final DRAE which yields the function's result, the annotated degree of this DRAE is the degree of the function the DRAC encodes. Before evaluating a DRAC, David has to verify that the degree of the polynomial it is supposed to evaluate equals the parametrized degree. Note that the definition of polynomial degree used in this thesis is different to the usual definition: For OPE, it is crucial that the evaluating party does not learn any coefficients of the polynomial because they can be regarded as the input values of the party which chooses the polynomial. Therefore the polynomial degree is the number of rows minus one of the vector a which specifies the polynomial ($\sum_{i=0}^{n} a_i \cdot x^i$). Hence, the degree of the polynomial $0 + 0x + x^2 + x^3 + x^4 + 0x^5 + 0x^6$ specified by $\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}^T$ is regarded as 6. This may seem odd but is not less powerful and does reveal leading coefficients of the encoding party. Obviously, if two parties agreed on polynomial degree n and the encoding party chooses to evaluate a polynomial of degree $m < n$ it can reshape the specification vector by trailing zeros until having $n + 1$ rows.

3 Correctness and Security of the Protocol

This chapter states and proves the security of the protocol in the *Universal–Composability* (UC) framework by Ran Canetti [Can01]. In the UC framework, security is defined by comparing an *ideal model* to a *real model*. The ideal model implements the intended functionality $\mathcal{F}$ safely by definition. The protocol under examination runs in the real model. In the real model, there is an adversary $\mathcal{A}$ that controls all corrupted parties. In the ideal model, there is a simulator $\mathcal{S}$ that tries to mimic $\mathcal{A}$. An environment $\mathcal{Z}$ is plugged to either the real or the ideal model and has to guess to which it is plugged to. A protocol Π is an universally composable implementation of the ideal functionality if, for every adversary $\mathcal{A}$ there is a simulator $\mathcal{S}$ such that for all environments $\mathcal{Z}$ the entire view of $\mathcal{Z}$ in the real model (with Π and $\mathcal{A}$) is statistically close to its view in the ideal model (with $\mathcal{F}$ and $\mathcal{S}$).

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z} : \text{ideal} \cong \text{real}$$

3.1 OPE Protocol & Functionality

This section formally defines the functionality $\mathcal{F}_{\text{OPE}}^{(\text{k,n})}$and the protocol Π_{OPE}this thesis proposes to realize OPE. The functionality described in Figures 3.1 and 3.2 is the ideal model of what the protocol tries to achieve. The protocol described in Figure 3.3 is a formalization of the OPE methodology as described in Section 2.6.

Functionality $\mathcal{F}_{\text{OPE}}^{(\text{k,n})}$

Parametrized by a finite field of size 2^k ($\mathbb{F}_{2^k}$) and polynomial degree n.

- Upon receiving input (`Commit`, a) from the environment, verify $a \in \mathbb{F}_{2^k}^{n+1}$ and that there is no stored input, yet; else ignore that input. Next record a.
- Upon receiving input (`Evaluate`, x) from the environment, verify $x \in \mathbb{F}_{2^k}$ and that there is no stored input, yet; else ignore that input. Next record x.
- Upon a and x are verified and stored, compute $y \leftarrow \sum_{i=0}^{n} a_i x^i$ and output (`Evaluated`, y).

Figure 3.1: The ideal $\mathbb{F}_{2^k}$-OPE functionality $\mathcal{F}_{\text{OPE}}^{(\text{k,n})}$

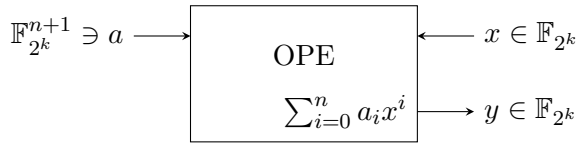

Figure 3.2: Graphical Representation of $\mathcal{F}_{\text{OPE}}^{(\text{k,n})}$

Protocol Π_{OPE}

Parametrized by the security parameter k that specifies the field $\mathbb{F}_{2^k}$ and the degree of the polynomial n.

Setup:

1. Upon input (`Commit`, a) from the environment, Goliath verifies that $a \in \mathbb{F}_{2^k}^{n+1}$; else ignores that input. Next, Goliath generates the DRAC and sets up the OAFE functionality for the polynomial $f(x) = \sum_{i=0}^{n} a_i x^i$ (see Section 2.6).
2. Finally, Goliath sends the DRAC to David. Goliath also allows David to use the OAFE functionality it configured before.

Evaluation:

1. Upon receiving the DRAC from Goliath, verify that there is no stored DRAC yet, else ignore that input. Next, David verifies the received input is a DRAC, else ignores that input. Additionally it verifies that the DRAC encodes a function whose degree is equal to n, else ignores the input. If the DRAC verification was successful, store the DRAC.
2. Upon receiving input (`Evaluate`, x) from the environment, verify that there is no stored value yet and that $x \in \mathbb{F}_{2^k}$, else ignore that input. If x has been verified successfully, store x as the value. If there is already a stored DRAC, continue at step 3, else wait for a stored DRAC.
3. Next, David evaluates the DRAC. Because the OAFE functionality is set up by Goliath, it is possible that during the evaluation the OAFE functionality turns out to behave unexpectedly: It could decline needed evaluations and it could return vectors of unexpected shapes. In both cases always assume it would have returned all zero vectors of the expected shape.
4. After the evaluation, David adds both components of the last DRAV and queries a special final OAFE with that value. Again, use the all zero vector if the OAFE behaves unexpectedly. The then received new value is the result of the computation $y = \sum_{i=0}^{n} a_i x^i$. Finally, output (`Evaluated`, y) to the environment.

Figure 3.3: Protocol: Oblivious Polynomial Evaluation

3.2 Simulators

This section describes two simulators which mimic some adversary $\mathcal{A}$ in the ideal model. There is a separate simulator for each of the potentially corrupted parties. The security proofs in Section 3.3 will use these simulators to proof the indistinguishability between the real model and the ideal model.

3.2.1 Simulator $\mathcal{S}_{\mathsf{David}}(\mathcal{A})$

Setup Phase:

- Setup an emulated version of the given real model adversary $\mathcal{A}$ which especially impersonates the corrupted David.
- Setup emulated, honest Goliath labeled $\mathcal{G}$.
- Setup emulated, honest OAFE functionality labeled $\mathcal{F}_{\text{OAFE}}$.
- Initialize $\mathcal{G}$ with a random polynomial of the parametrized degree k and record the DRAC C it generated.
- Wire $\mathcal{G}$, $\mathcal{F}_{\text{OAFE}}$ and $\mathcal{A}$ to each other and $\mathcal{A}$ to the environment as in the real model.
- Start the simulation by transitioning to the *Processing Phase*.

Processing Phase:

- Intercept the adversary's ($\mathcal{A}$) polynomial input x. This is trivial because it is the first message that $\mathcal{A}$ transmits to $\mathcal{F}_{\text{OAFE}}$ (only considering messages accepted by $\mathcal{F}_{\text{OAFE}}$). Notice, this is the initial DRAV encoding of the plain value x. Next, halt the simulation and evaluate the proper polynomial using the intercepted input x and the ideal functionality $\mathcal{F}_{\text{OPE}}$ by sending it (`Evaluate`, x).
- Upon receiving (`Evaluated`, y) from $\mathcal{F}_{\text{OPE}}$, calculate the value χ with which an honest David would evaluate the last OAFE . Because an honest David acts entirely deterministic, this calculates from the stored DRAC C, the OAFE parameters in $\mathcal{F}_{\text{OAFE}}$, and the stored input x. Next, draw a random value r and calculate $\psi = \frac{y-r}{\chi}$. Next, reconfigure the parameters of the last OAFE to (ψ, r). The last OAFE then calculates the affine function $f(\eta) = \psi \cdot \eta + r$ which—when evaluated with χ—calculates the result y of the proper polynomial at the node x. Since any honest David evaluates the last OAFE with χ, it will receive the correct result $y = f(\chi) = \psi \cdot \chi + r = \frac{y-r}{\chi} \cdot \chi + r = y$. Any dishonest David will receive some random value depending on the random value r. Finally unhalt the simulation, no further interceptions are necessary.

3.2.2 Simulator $\mathcal{S}_{\mathsf{Goliath}}(\mathcal{A})$

The setup phase is run as soon as the simulator starts.

Setup Phase:

- Setup emulated, honest David and OAFE functionality $\mathcal{F}_{\text{OAFE}}$.
- Wire the given real model adversary $\mathcal{A}$ that impersonates Goliath with the emulated David and $\mathcal{F}_{\text{OAFE}}$. Additionally wire $\mathcal{A}$ with the environment the way they would be wired in the real model.

Processing Phase:

- Upon the emulated OAFE functionality received the OAFE configuration and the emulated David received the DRAC, symbolically evaluate the polynomial that $\mathcal{A}$ described. During the symbolic evaluation, the OAFE functionality configuration could proof to be unfitting: $\mathcal{A}$ could have sent a configuration that allows too few OAFE evaluations or evaluates to vectors of wrong shape. In both cases, modify the configuration to return all zero vectors of the correct shape whenever the DRAC evaluation would fail otherwise.
- After having symbolically evaluated the polynomial encoded by $\mathcal{A}$, verify, that the polynomial has a degree that equals the parametrized degree k. If $\mathcal{A}$ submitted a polynomial of valid degree, store the polynomial's coefficients as $a \in \mathbb{F}_{2^k}^n$. Next, upload that polynomial to the ideal functionality $\mathcal{F}_{\text{OPE}}$ by sending (`Commit`, a) to $\mathcal{F}_{\text{OPE}}$, the simulation is now terminated, wait for the adversary to terminate as well. If the polynomial was illegal, restart the simulator in the setup phase. This restart is equivalent to ignoring the adversary's inputs.

3.3 Security Proof

The proof is a distinction between the two relevant settings: Either Goliath or David is corrupted. The proof below handles theses cases separately. The remaining possibilities that either no party is corrupted or both parties are corrupted do not have to be proved separately. No meaningful proof is possible if both parties are corrupted and no proof is necessary if all involved parties act honest.

3.3.1 Corrupted David

This section will proof the security of the proposed protocol Π_{OPE}(Figure 3.3) in the Universal Composability (UC) framework [Can01] against both, passive and active adversaries impersonating David. Passive adversaries are adversaries which try to calculate additional information from intermediate results, active adversaries additionally misuse the protocol in some unpredictable way.

The protocol is trivially secure against passive adversaries impersonating David because every value that gets transmitted to David is one–time encrypted, except the very last one which is the plain result of the evaluated polynomial. The values sent to David are part of a DRAE and DRAEs only contain references to OAFEs and DRAVs. Section 2.3 proves (Lemma 1) that DRAVs only transport uncorrelated uniform randomness to parties not in possession of the encryption keys. The OAFE evaluations yield further (one–time encrypted) DRAVs and radicals (see Section 2.4.2.2) which are one–time encrypted as

well. The last OAFE only yields the unencrypted result which David should obtain anyways and therefore is uninteresting. The simulator from Section 3.2.1 plays on this property and encodes a random polynomial of the correct degree. Neither David nor the environment will not be able to distinguish the random polynomial from any other polynomial of the same degree because everything they might learn is one–time pad encrypted. Therefore, the protocol is perfectly secure against any passive adversary.

For the security against an actively corrupted David, an hybrid argument is employed. The general idea is to transform an actively corrupted adversary incrementally into a passively corrupted adversary and to show that the statistical distance in the environment's view is negligible. For two random variables x and y, the statistical distance Δ_S is denoted using the following standard notation.

$$\Delta_S(x, y) = \frac{\sum_\alpha |Pr[x = \alpha] - Pr[y = \alpha]|}{2}$$

The first step is to attach a *monitor module* between the adversary $\mathcal{A}$ and the OAFE functionality $\mathcal{F}_{\text{OAFE}}$. The monitor module is parametrized by a transcript of how an honest David would act. The task of the module is to analyze the messages that David (impersonated by the adversary) sends to the OAFE functionality. Upon the detection of a message which does not match the honest transcript, it changes this message to match the honest message. Additionally the monitor module intercepts the response to the changed message (sent by $\mathcal{F}_{\text{OAFE}}$) and changes the response to some value uniformly at random. The monitor module does this change only once. If the adversary changes one of the following messages, the monitor module does no further changes. The union of the adversary $\mathcal{A}$ and the monitor module can be seen as a transformed adversary $\mathcal{A}$' which is forging (at the earliest) the message following the first message that $\mathcal{A}$ illicitly changed. So, if $\mathcal{A}$ forges n messages, $\mathcal{A}$' forges at most $n - 1$ messages.

The interesting part is the statistical distance of the environment's view between $\mathcal{A}$ and $\mathcal{A}$'. The message which $\mathcal{A}$ will receive from the monitor module after having forged a message for the first time is uniform randomness. This will render some DRAV that is computed involving value in the message to a non–well–formed DRAV with overwhelming probability. The probability equals $1 - 1/|\mathbb{K}| = 1 - 1/2^k$ because exactly one element of the finite field $\mathbb{K}$ matches its counterpart tuple component (see Section 2.3). Therefore, the statistical distance $\delta_{\mathcal{A},\mathcal{A}'}$ of the environment's view between $\mathcal{A}$ and $\mathcal{A}$' is:

$$\delta_{\mathcal{A},\mathcal{A}'} = \Delta_S(\text{view}_{\mathcal{Z}}(\mathcal{A}), \text{view}_{\mathcal{Z}}(\mathcal{A}')) \leq \frac{1}{|\mathbb{K}|} = \frac{1}{2^k}$$

This argument can be used inductively until the original adversary $\mathcal{A}$ is transformed into a passive adversary $\mathcal{A}^*$ which does not illicitly change any message anymore. For example, if the original adversary $\mathcal{A}$ forges three messages, then $\mathcal{A}'$ ($\mathcal{A}$ plus monitor module) forges two, $\mathcal{A}''$ ($\mathcal{A}'$ plus monitor module) forges one, and $\mathcal{A}'''$ forges no messages. Per definition, $\mathcal{A}'''$ is a passively corrupted adversary. Because the triangle inequality holds for the statistical distance and the maximal number of OAFEs (and therefore the maximal number of messages the adversary can forge) used to evaluate a polynomial of degree n is in $O(n)$, the statistical distance between any active adversary $\mathcal{A}$ and some passive adversary $\mathcal{A}^*$ is

$$\Delta_S(\text{view}_{\mathcal{Z}}(\mathcal{A}), \text{view}_{\mathcal{Z}}(\mathcal{A}^*)) \leq \frac{O(n)}{2^k}$$

And because k is the security parameter, the statistical distance in the environment's view between any active adversary and any passive adversary is negligible. As stated above, the protocol is perfectly secure against any passive adversary and therefore information theoretically UC–secure against any active adversary. □

3.3.2 Corrupted Goliath

A corrupted Goliath has the following possibilities to cheat:

1. Describe a polynomial with a degree other that the parametrized degree k.
2. Configure the OAFE functionality in a way that the DRAC evaluation would fail because it assumes additional evaluation possibilities or vectors of other shapes.
3. Send messages that do not describe a valid DRAC.

Both, in the ideal model (running the simulator from 3.2.2) as well as in the real model (running the Π_{OPE} (Figure 3.3) and the adversary), these possibilities are handled indistinguishably for any environment:

1. The protocol and the simulator verify the polynomial's degree. Both ignore the input when the degree is other than parametrized.
2. If the OAFE functionality is configured in a wrong way, both handle this case similarly: Vectors of wrong shapes are turned to the all zero vector of the correct shape. Missing OAFE evaluations are assumed to return the all zero vector of the expected shape.
3. Messages that do not validly describe a DRAC are ignored.

This assures that any environment will not be able to distinguish between the real and the ideal model for any adversary using the simulator from Section 3.2.2. □

4 Implementation

In addition to the formal methodology (Chapter 2) and the security proof of the protocol in the UC framework (Chapter 3), this thesis contains an implementation of the protocol. The implementation is written in the lazy, functional programming language Haskell[1], more specifically Haskell as implemented by the major Haskell compiler, the Glasgow Haskell Compiler[2] (GHC) version 7.6.1. Whenever feasible, standard *Haskell 2010* [M+] features have been preferred. However, the following extensions have been used: `BangPatterns`, `ScopedTypeVariables`, `Rank2Types`, and `FlexibleContexts`. The Protocol Buffers (see Chapter 4.3.5) serialization and parsing code that is generated automatically by `hprotoc`[3] additionally uses `DeriveDataTypeable`, `FlexibleInstances`, and `MultiParamTypeClasses`. The code should therefore be compilable by any Haskell compiler that supports these extensions and is able to compile the external libraries.

The implementation is designed to match the descriptions in chapter 2 as closely as possible, it implements Oblivious Polynomial Evaluation as explained in Chapter 2.6.

Even the Haskell data–types such as `DRAC`, `DRAE`, `RAE`, `LinearExpr` match the names of the constructs described in chapter 2: DRACs (Chapter 2.5), DRAEs (chapter 2.4) and linear expressions.

As in Chapter 2.6, the oblivious polynomial evaluation consists of three parties: The OAFE issuing party Goliath (executable program called `Goliath`), the receiving party David (`David`) and the OAFE functionality (`Token`). The individual programs talk to each other using TCP/IP networking. Even though `Token` implements the OAFE functionality $\mathcal{F}_{\text{OAFE}}$, it does not implement the David & Goliath protocol [DKMQ12] but only its interface. A real implementation using a tamper–proof hardware token is left open to potential future work.

4.1 Communication Channels

The binaries `Goliath`, `David`, and `Token` use TCP/IP networking to communicate. In the following, the different TCP ports and the communication which takes part over these channels are described.

Goliath to Token

On TCP port `23120` Goliath initiates a connection to the Token. This connection serves to transfer the OAFE configuration. After having successfully received the OAFE configuration, the OAFE functionality accepts OAFE evaluation requests (usually from David) but no more OAFE configurations. One Token process can be configured to a OAFE configuration exactly once.

[1] `http://haskell.org`
[2] `http://www.haskell.org/ghc/`
[3] `http://hackage.haskell.org/package/hprotoc`

David to Token

On TCP port `23021` David initiates a connection to the Token. This connection serves to evaluate the OAFEs. David tells which OAFE to evaluate and the value. The Token then replies with a vector consisting of the values of the evaluated linear expressions.

David to Goliath

On TCP port `23201` David initiates a connection to Goliath. This connection is not necessary for the protocol itself but serves to exchange settings between Goliath and David. Goliath tells David the host name and the TCP port of the Token and David tells Goliath the port on which David accepts the DRAC.

Goliath to David

On TCP port `23102` Goliath initiates a connection to David. This connection streams the DRAC that David will evaluate.

4.2 Differences Between the Implementation and this Writing

The main difference between the current implementation and this writing is that the OAFE functionality is not implemented using the David & Goliath [DKMQ12] protocol. Instead, the `Token` implements a naive implementation of the ideal OAFE functionality. Further works could change the implementation to a real implementation of the protocol. Another difference is that David does not verify the polynomial degree it evaluates.

4.3 Implementation Details

This section describes implementation details that could be of interest or need documentation.

4.3.1 Calculations in Finite Fields

For calculations in finite fields, the C++ library NTL[4] (*Number Theory Library*) has been interfaced to Haskell to use in the implementation of this thesis. The interface has been developed using Haskell's *Foreign Function Interface* (FFI) [M+] and *C* $\longrightarrow$ *Haskell* (C2Hs) [Cha00]. Before interfacing the library to Haskell a very lightweight C wrapper has been developed because the library is written in C++ which cannot directly be interfaced by the FFI.

Although NTL does its work fast and without problems, the library has some implementation problems and is not optimally designed for exposing its functionality to functional programming languages. First, the library is neither thread–safe nor reentrant because it globally stores internal state that can be modified using library functions. The small C wrapper library therefore tries to provide a thread–safe interface to NTL by limiting its functionality. Another problem is that NTL is designed for mutable objects, for example, arithmetic operations are implemented using destructive assignment. Since Haskell is a

[4] `http://www.shoup.net/ntl/`

pure functional language [M+] and destructive assignment is referentially opaque, the C wrapper changes that and offers only an interface that obeys referential transparency. It does so by internally allocating a new element for the result of arithmetic operations and other functions that would modify existing objects otherwise.

The interface that obeys referential transparency makes the binding work well inside Haskell programs. But for large number of NTL objects, there is a performance problem: Creating and destructing a massive amount of objects is slow. The problem is that it is not possible to allocate memory for NTL objects completely externally because NTL allocates memory internally on the heap in the constructor. In more detail: The objects the programmer gets from the NTL library are only one machine word wide which stores a pointer to memory on the heap which stores the representation of the elements in the finite field. In other words: Allocating memory for an array of field elements is not in $O(1)$ as usual (a chunk of stack space or one `malloc()` call) but in $O(n)$ because NTL allocates additional memory for every element on the heap internally. Obviously, the same problem occurs when destructing an array of NTL objects: Usually that is in $O(1)$ as well (waiting until the stack frame gets destroyed or `free()` is called) but using NTL this is in $O(n)$ because C++'s `delete` has to be called for every element. Not calling `delete` will lead to a massive memory leak. Therefore, the Haskell binding cannot manage memory as usual but has to call a special memory freeing function of the C wrapper that calls C++'s `delete` for every object. This becomes a major performance problem for evaluations of large polynomials (see chapter 5.2). Nevertheless, this problem is easily fixable by using a Haskell library for finite field calculations or interfacing a library that fits better.

To benchmark this thesis (see Chapter 5), a second library (in $\mathbb{F}_{97}$, from the `HaskellForMaths`[5] package) has been used, this library does not have the same problems as NTL.

4.3.2 Thread Safety

The implementation itself is fully thread–safe, all communication is done using GHC's implementation of *Software Transactional Memory* (STM) [HMPJH05] for Haskell. The library (NTL) that is used for the calculations in the finite fields is not thread–safe but the small C wrapper (see Chapter 4.3.1) tries to fix NTL's thread–safety problems. However, for guaranteed thread–safety NTL should be replaced by another library that is thread–safe. The NTL implementation also has other disadvantages that would make another library even more useful (for details see Chapter 4.3.1).

4.3.3 Testing the Correctness of the Implementation

The correctness of the implementation has not been proofed formally since that is not feasible for complex programs. Instead of a real proof, the implementation has been tested using the specification–driven automatic Haskell testing tool QuickCheck [CH00] and the classic unit test approach of HUnit[6].

[5] `http://hackage.haskell.org/package/HaskellForMaths`
[6] `http://hunit.sourceforge.net/`

4.3.4 Cryptographic Randomness

The random numbers needed for the implementation of this thesis are generated using `monadcryptorandom`[7]. This package allows explicit failure of the random number generation and exchangeable random number generators such as `DRBG`[8]. `DRBG` implements a NIST standardized number–theoretically secure random number generator.

4.3.5 Network Communication Layer

The communication between the different binaries is done using the schrieblesque Google™ Protocol Buffers[9]. This library features easy network communication with complex data types independent of the programming language, operating system, machine endianness, and other parameters. The Haskell Protocol Buffers compiler `hprotoc` that comes with the Haskell Protocol Buffers library (`protocol-buffers`[10]) automatically generates the Haskell source code which serializes the data structures to Protocol Buffers, and code parses Protocol Buffers to Haskell data structures. The automatically generated code can be found in the directory `gen-src/` in this thesis' source tree.

4.3.6 Other Libraries

This thesis uses many other libraries mainly from HackageDB[11]. The complete list of packages can be extracted from the file `diplomarbeit.cabal` that comes with the source code of this thesis.

4.3.7 Source Tree Organization

A listing of the most important directories in the source code tree of this thesis can be found below.

- `programs/`, source code for the binaries (`Goliath`, `David` and `Token`).
- `lib/`, source code for the library functionality.
- `lib/Functionality/`, reusable code that forms the binaries' functionality.
- `lib/Data/OAFE/`, OAFE functionality implementation.
- `lib/Data/RAE/`, RAE encoding, decoding and evaluation.
- `lib/Math/`, finite field implementation and other mathematical functions.
- `test/`, QuickCheck tests and the unit tests.
- `bench/`, benchmarking programs.
- `scripts/`, helper scripts, mainly for benchmarking and testing.

[7] `http://hackage.haskell.org/package/monadcryptorandom`
[8] `http://hackage.haskell.org/package/DRBG`
[9] `http://code.google.com/apis/protocolbuffers/`
[10] `http://hackage.haskell.org/package/protocol-buffers`
[11] `http://hackage.haskell.org/packages/hackage.html`

- `protos/`, Protocol Buffers description files.
- `gen-src/`, automatically generated source code, derived from the Protocol Buffers descriptions files.
- `writings/`, this writing.
- `evaluation/`, data for the figures of this writing.
- `dist/`, after having compiled the source code, this folder contains the binaries.

4.3.8 Build System and Building

To build the source code of this thesis, Cabal[12] is used. Cabal is the standard building and packaging system used for Haskell programs and libraries. The build process can be started by executing the following commands in a shell:

```
$ cd diplomarbeit/
$ cabal install --only-dependencies   #install required packages
$ cabal configure
$ cabal build
```

4.4 User Input/Output

Whenever field elements in large fields that are no prime fields (e.g., $\mathbb{F}_{2^{256}}$) are read from user input or are outputted to the user, a special text format is used. Note that scalar field elements of non–prime fields are elements from polynomial rings and therefore written as polynomials. This implementation uses the following format (given as a regular expression) to denote the polynomials (that represent the scalar field elements):

```
\[([01]( [01]){0,255})?\]
```

Intuitively that is between 0 and 256 digits (each either `0` or `1`) separated by spaces and surrounded by square brackets. The meaning of such a string is $\sum_{p=0}^{P} d_p \cdot x^p$ where d_p is the digit at index p in the string and P is the maximal index (counted from 0, not counting the spaces and the brackets). An exception is the string `[]` which represents the polynomial `0`. Examples:

- `[]` and `[0]` represent 0
- `[1]` represents $1 \cdot x^0 = 1 \cdot 1 = 1$
- `[0 1]` represents $0 + 1 \cdot x^1 = x$
- `[1 0]` represents $1 + 0 \cdot x^1 = 1$
- `[1 1]` represents $1 + 1 \cdot x^1 = 1 + x$
- `[1 0 1 0 1 0]` represents $1 + x^2 + x^4$

The default implementation uses $\mathbb{F}_{2^{256}}$ specified by the irreducible polynomial $1 + x^2 + x^5 + x^{10} + x^{256}$. $\mathbb{F}_{97}$ is a prime field and therefore the elements are regular elements of the ring $\mathbb{Z}/2\mathbb{Z}$. This implementation uses regular numbers for any prime field. Exemplary for $\mathbb{F}_{97}$: $1 \hat{=} 1(mod 97) = 1_{\mathbb{F}_{97}}$, $98 \hat{=} 98(mod 97) = 1_{\mathbb{F}_{97}}$.

[12] `http://www.haskell.org/cabal/`

4.5 Usage of the Programs

The three main programs are `Goliath`, `David` and `Token`. `Token` does not accept any command line parameters. Goliath expects exactly one command line parameter: The file with the polynomial to evaluate, one coefficient per line. David expects exactly one parameter too: The field element to evaluate the polynomial with. The following example would evaluate the polynomial $f(x) = a \cdot 1 + b \cdot x + c \cdot x^2$ where $a = 1 \stackrel{\wedge}{=}$ `[1]`, $b = 1 + p^2 \stackrel{\wedge}{=}$ `[1 0 1]`, $c = (1 + p^2 + p^4) \stackrel{\wedge}{=}$ `[1 0 1 0 1]`, and $x = 1 + p^2 + p^3 + p^4 + p^6 \stackrel{\wedge}{=}$ `[1 0 1 1 1 0 1]`:

```
$ (echo [1]; echo [1 0 1]; echo [1 0 1 0 1]) > /tmp/test.poly
$ Goliath /tmp/test.poly
$ Token
$ David '[1 0 1 1 1 0 1]'
```

The same procedure can be run by the provided `run-evaluation.sh` script that comes with this thesis:

```
$ run-evaluation.sh /tmp/test.poly '[1 0 1 1 1 0 1]'
```

Additionally, there is an implementation of the procedure in only one binary, that *does not* use TCP/IP networking. This optional version can be used in the following way:

```
$ AllInOne /tmp/test.poly '[1 0 1 1 1 0 1]'
```

4.6 Documentation

The implementation is documented using Haddock[13] the standard tool for automatically generating documentation from annotated Haskell source code. The generated HTML documentation for the implementation of this thesis is available at:

`http://johannesweiss.eu/diplomarbeit/index.html`

4.7 Code Availability

All of the code is open–sourced under the terms of the *GNU General Public License (Version 3)*[14] and available at GitHub[15]. The code can be obtained by executing the following command in a shell:

```
$ git clone https://github.com/weissi/diplomarbeit.git
```

[13] `http://www.haskell.org/haddock/`
[14] `http://www.gnu.org/licenses/gpl-3.0.html`
[15] `https://github.com/weissi/diplomarbeit.git`

5 Evaluation

This part evaluates the computational complexity of the methodology and the performance of the implementation.

5.1 Test Setup

5.1.1 Test Machines

Both test machines are regular home computer systems, thus the benchmarks were not run under laboratory conditions, Though most background services and other programs that could interfere with the benchmarks have been stopped. The implementation has been compiled using full optimizations turned on (`$ cabal configure --ghc-options=-O3`). To work out the computing time taken for the implementation of the protocol itself, the protocol implementation is evaluated using two different finite field implementations. The first implementation is a C++ library (see Section 4.3.1) for calculation in $\mathbb{F}_{2^{256}}$ and the second implementation is a Haskell library which calculates in the prime field $\mathbb{F}_{97}$.

Test Machine 1

Dell Latitude D620 running *Debian GNU/Linux* on kernel version *3.1.7, 32–bit*. The machine is powered by a *Intel® Core Duo T2400* dual core processor at *1.83 GHz*. It has *2 GB* of system memory.

Test Machine 2

Apple Mac mini (`Macmini 3,1`) running *MacOS X 10.7.5 (Lion), 64–bit*. The machine is powered by a *Intel® Core 2 Duo* dual core processor at *2.26 GHz*. It has *8 GB* of system memory.

5.1.2 Benchmarking Process

Both machines run the three programs (`Goliath`, `David`, `Token`) simultaneously, the programs are compiled single–threaded. They communicate with each other using local TCP/IP networking. Section 4.1 has an in–depth description about the communication that takes part in the implementation that comes with this thesis. The running time includes the DRAC building and transmitting, the OAFE configuration and the successful evaluation of a random polynomial. The polynomial is evaluated using *Horner's rule* [CLRS01]. The data points in all figures and tables were obtained by taking the median of running the same benchmark five times in a row (preceded by a non–accounted warm–up run).

5.2 Computational Complexity

Ordinary evaluation of a polynomial of degree n using *Horner's rule* is in $\Theta(n)$ [CLRS01]. Despite the complex technique presented in this thesis the overall evaluation time is still in $\Theta(n)$. This is true for both, the theoretical examination and the implementation. Figure 5.1 and Table 5.2 show this for the finite field $\mathbb{F}_{97}$ ($\mathbb{F}_{97}$ implementation from the `HaskellForMaths`[1] package). The implementation using the field $\mathbb{F}_{2^{256}}$ does not seem to have linear time complexity. This is caused by the suboptimal implementation and the Haskell binding of the library used for the calculations in $\mathbb{F}_{2^{256}}$. It is problematic that the library includes a rather complex routine for object destruction that takes most of the evaluation time. Root of the problem is that functional languages prefer immutable (unchangeable) objects with very fast construction/destruction and the C++ library that is interfaced is optimized for as few constructions/destructions as possible and the use of mutable objects. The phenomenon that is observable when using this library is that the garbage collector is taking up most of the running time (up to 85%) waiting for the library destruction routine. For high performance (such as the pure Haskell implementation for $\mathbb{F}_{97}$) another library should be interfaced or developed entirely in Haskell, but this is not the main concern of this thesis. Further details are covered in Section 4.3.1. For lower degrees, the implementation shows its linear time complexity even when used with the suboptiomal $\mathbb{F}_{2^{256}}$ implementation, Figure 5.2 and Table 5.2 manifest this. It is also noteworthy that the first test machine (see Section 5.1.1) outperforms the second test machine despite the inferior hardware. The cause has not been investigated, but a possible cause could be that the 64–bit GHC runtime seems to be slower (at least on Apple Macs) [Len11].

5.3 Memory Complexity

The memory complexity of the current implementation is in $O(n)$, n being the polynomial's degree, too. The reason is that the implementation has to store all DRAEs to be able to generate the OAFE configuration and to write the OAFE references in the DRAEs.

[1] `http://hackage.haskell.org/package/HaskellForMaths`

Polynomial Degree	Running Time $\mathbb{F}_{97}$ [s]	Running Time $\mathbb{F}_{2^{256}}$ [s]
500	1.754	2.389
1000	3.673	6.032
1500	5.587	10.159
2000	7.400	15.179
2500	9.104	20.702
3000	10.905	25.560
3500	12.800	31.978
4000	14.549	39.926
4500	16.422	47.248
5000	18.288	55.450
5500	20.142	63.496
6000	22.000	72.293
6500	23.901	85.184
7000	25.654	93.694
7500	27.670	104.231
8000	29.510	117.480
8500	31.518	127.848
9000	33.417	143.123
9500	35.525	156.820
10000	37.395	172.216

Table 5.1: Evaluation Time of Polynomials by Degree on Test Machine 1

Polynomial Degree	Average Running Time [ms]
0	70
50	219
100	393
150	588
200	779
250	1002
300	1225
350	1483
400	1718
450	1964
500	2294

Table 5.2: Evaluation Time of $\mathbb{F}_{2^{256}}$ Polynomials (Lower Degrees) on Test Machine 1

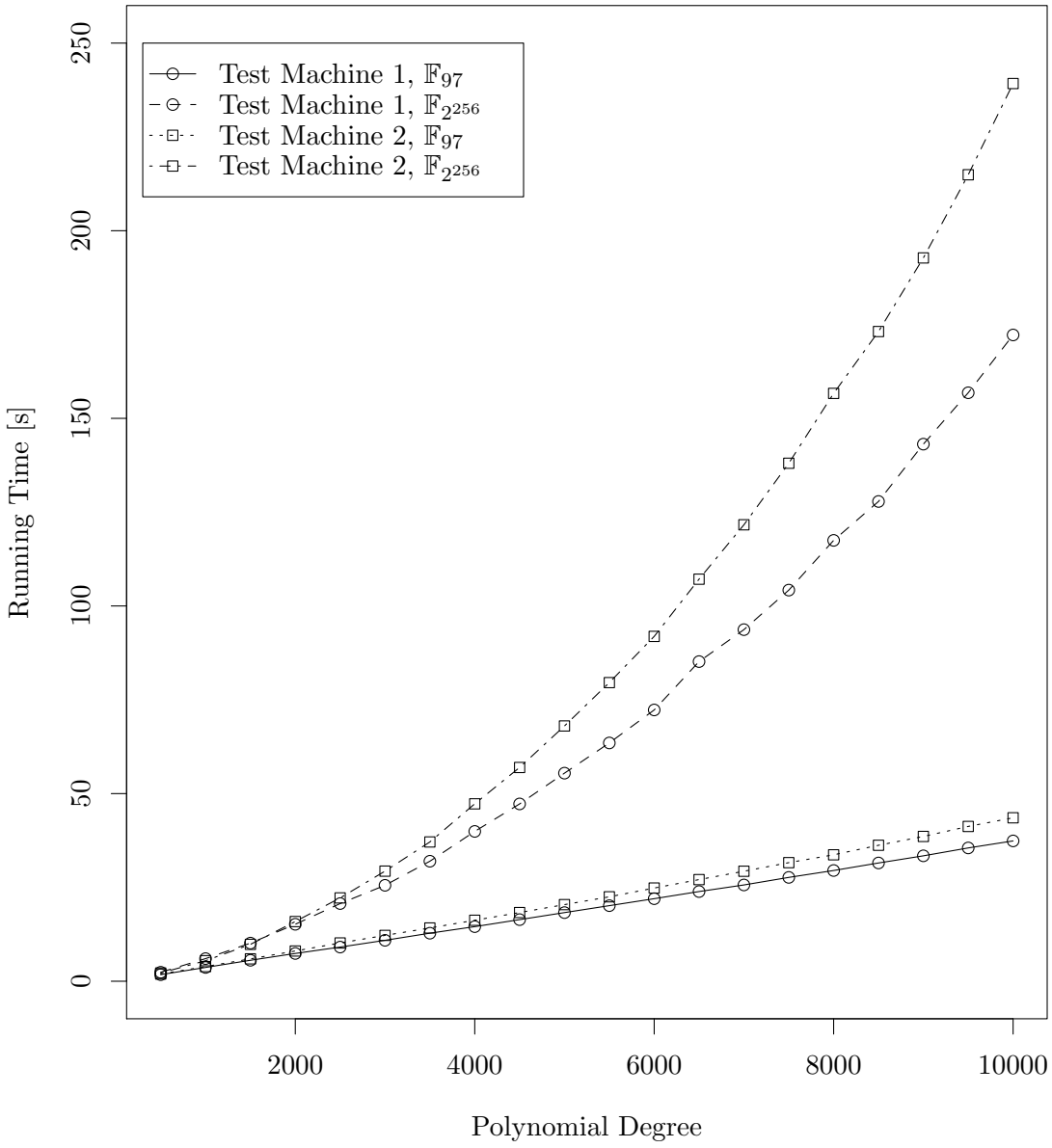

Figure 5.1: Evaluation Time of Polynomial by Degree

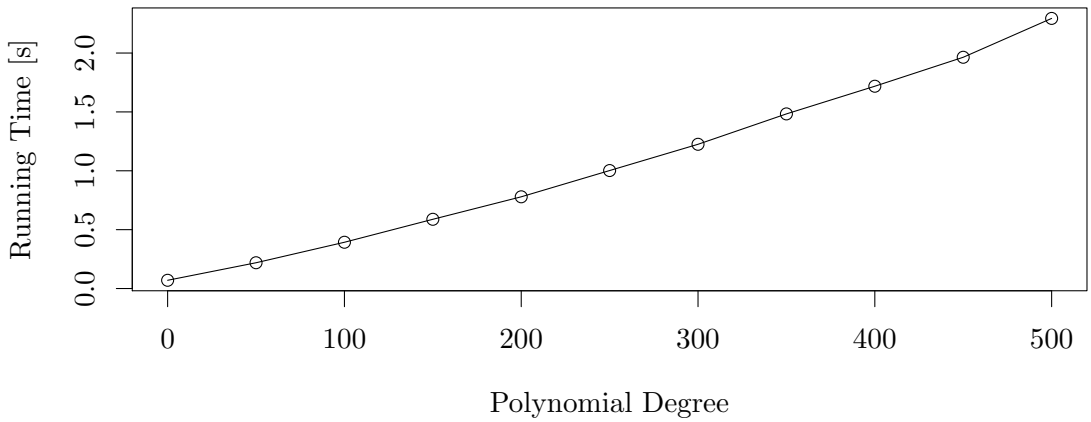

Figure 5.2: Evaluation Time of $\mathbb{F}_{2^{256}}$ Polynomials (Lower Degrees) on Test Machine 1

6 Conclusion

This thesis contributes a protocol implementing *Oblivious Polynomial Evaluation* in linear time. The protocol is implemented and proven to be UC–secure [Can01] against passively and actively corrupted parties. Hence, the protocol is ideally suited as a cryptographic primitive to build future work on.

For restricted security requirements, evaluation of arbitrary arithmetic circuits (including Square & Multiply [Knu81]) is possible using the techniques presented. More specifically, restricted security requirements means security against an actively corrupted evaluating party (David) but security against a passively corrupted function definition party (Goliath) only.

6.1 Open Questions and Outlook

The most important open question is to find an improvement for the current technique to support arbitrary arithmetic circuits and not just polynomials while maintaining the computing time complexity and the UC–security against all actively corrupted parties.

The implementation would benefit from a better implementation of a library for calculations in large finite fields such as $\mathbb{F}_{2^k}$ for $k > 128$. Additionally, the David & Goliath protocol [DKMQ12] should be implemented as a component the implementation of this thesis could use. The current implementation assumes a tamper–proof hardware token that implements the OAFE functionality (the program `Token` that comes with this thesis implements the functionality naively but insecure).

7 Discontinued Approaches

This chapter briefly describes the approaches that have been investigated (and partly implemented) but were considered not good enough to reach the goal. The approaches are ordered chronologically by date of examination to document the evolution of the methodology.

7.1 Linear Bijection Straight–Line Programs

The first approach pursued in this thesis to transform general formulas to affine functions suitable for OAFEs was via Linear Bijection Straight–Line Programs. The results seemed promising at the first glance but a problem emerged when using multiplications. The problem already becomes apparent as Cleve writes in the abstract of his paper [Cle91]:

> We show that, over an arbitrary ring, for any fixed $\epsilon > 0$, all balanced algebraic formulas of size s are computed by algebraic straight–line programs that employ a constant number of registers and have length $O(s^{1+\epsilon})$.

Two problems arise: Cleve's approach does not support *Square and Multiply* and it is slightly polynomial. However, the approach is partly implemented as explained in detail in the next few sections but neither the code nor the ideas are used for the final version of this thesis.

7.1.1 From Arithmetic Formulas to Matrix Multiplications

The definition of formulas is the same as Cleve's [Cle91]: Formulas are circuits that are trees. A postorder traversal is enough to evaluate the formula. This section describes the evaluation of such a formula using *linear bijection straight–line programs* (LBS programs) [Cle91] which use at most ω registers. An LBS program can be simulated by matrix multiplications, one statement is simulated by one matrix multiplication. The matrices are elements of $SL_w(K)$, the special linear group consisting of $\omega \times \omega$ matrices with determinant 1 (and K a field).

A LBS program consists of assignment statements of the following forms where $R_{1,\ldots,\omega}$ denote registers, $c \in K$ constants and $x_u \in K$ the formula's inputs:

$$
\begin{aligned}
R_j &\leftarrow R_j + (R_i \cdot c) \\
R_j &\leftarrow R_j - (R_i \cdot c) \\
R_j &\leftarrow R_j + (R_i \cdot x_u) \\
R_j &\leftarrow R_j - (R_i \cdot x_u)
\end{aligned}
$$

7.1.1.1 Transformation of Formulas to LBS Programs

The goal is to transform a register R_{out} with a initial value of 0 to be transformed like $R_{out} \leftarrow R_{out} + R_{one} \cdot f(x_G, x_D)$. The special register R_{one} holds a constant 1. This can be achieved by induction as follows. For the exact definitions, proofs and algorithms how to transform arbitrary formulas to LBS programs, see [Cle91].

Depth $d = 0$ The construction of the LBS for $d = 0$ is straightforward: $R_j \leftarrow R_j \pm R_i \cdot c$ or $R_j \leftarrow R_j \pm R_i \cdot x_u$.

Depth $d > 0$ Two LBS programs that have the effect of $R_j \leftarrow R_j \pm R_i \cdot l(x_G, x_D)$ and $R_j \leftarrow R_j \pm R_i \cdot r(x_G, x_D)$ express formulas of depth $d > 0$ using only formulas of depth $d - 1$. Repeated until $d = 0$, arbitrary formulas can be written as LBS programs. The following two sections show the induction step for additions and multiplications.

Additive: The construction of an LBS program doing $R_j \leftarrow R_j + R_i \cdot (l + r)(x_G, x_D)$ can be achieved by the following LBS program.

$$R_j \leftarrow R_j + R_i \cdot l(x_G, x_D)$$
$$R_j \leftarrow R_j + R_i \cdot r(x_G, x_D)$$

Alike for $R_j \leftarrow R_j - R_i \cdot (l + r)(x_G, x_D)$

$$R_j \leftarrow R_j - R_i \cdot l(x_G, x_D)$$
$$R_j \leftarrow R_j - R_i \cdot r(x_G, x_D)$$

Multiplicative: The construction of an LBS program doing $R_j \leftarrow R_j + R_i \cdot (l \cdot r)(x_G, x_D)$ is less obviously achieved by the following LBS program.

$$R_k \leftarrow R_k - R_j \cdot r(x_G, x_D)$$
$$R_j \leftarrow R_j + R_i \cdot l(x_G, x_D)$$
$$R_k \leftarrow R_k + R_j \cdot r(x_G, x_D)$$
$$R_j \leftarrow R_j - R_i \cdot l(x_G, x_D)$$

Alike for $R_j \leftarrow R_j - R_i \cdot (l \cdot r)(x_G, x_D)$

$$R_k \leftarrow R_k - R_j \cdot r(x_G, x_D)$$
$$R_j \leftarrow R_j - R_i \cdot l(x_G, x_D)$$
$$R_k \leftarrow R_k + R_j \cdot r(x_G, x_D)$$
$$R_j \leftarrow R_j + R_i \cdot l(x_G, x_D)$$

7.1.1.2 Implementation State

The current implementation is a Haskell module which features the transformation from an arithmetic expression to an LBS program. The LBS program then transforms easily to the matrices. The multiplication of the matrices yield the result. The implementation can be found in the file `lib/Codec/LBS.hs` of the code tree of this thesis (see Section 4.3.7 and 4.7). Exemplary, the LBS program to evaluate the function $f(x_G, x_D) = 3x \cdot (x_G + x_D^2)$ which will hold the result in `R1`:

```
R1 <- R1 - R2 * Xg
R1 <- R1 - R3 * Xd
R3 <- R3 - R2 * Xd
R1 <- R1 + R3 * Xd
R3 <- R3 + R2 * Xd
R2 <- R2 - R3 * Xg
R3 <- R3 + R0 * 3
R2 <- R2 + R3 * Xg
R3 <- R3 - R0 * 3
R1 <- R1 + R2 * Xg
R1 <- R1 - R3 * Xd
R3 <- R3 + R2 * Xd
R1 <- R1 + R3 * Xd
R3 <- R3 - R2 * Xd
R2 <- R2 - R3 * Xg
R3 <- R3 - R0 * 3
R2 <- R2 + R3 * Xg
R3 <- R3 + R0 * 3
```

The construction of the matrices is straightforward: The statement $R_i \leftarrow R_i + (R_j \cdot \alpha)$ is equivalent to the $K^{\omega \times \omega}$ identity matrix whose entry i, j is set to α.

7.1.2 Grouping the Matrices

The grouping process is straightforward: From the process described in Section 7.1.1 matrices $\widehat{M_1}$ to $\widehat{M_n}$, which each have the effect of exactly one LBS program statement, are obtained. Using associativity, groups of a variable amount of these matrices can be built. Each group is complete when there is at least one reference to the *other party's input* x_D. The matrices M_1 to M_m are the result of this step. The following properties hold:

$$n \geq m$$
$$\prod_{i=1}^{m} M_i = \prod_{j=1}^{n} \widehat{M_j}$$

7.1.3 Garbling the Matrices

Let D_L be the $\omega \times \omega$ matrix whose entry $2, 2$ is 1, and whose other entries are 0. Multiplication of D_L selects the second row of matrices multiplied on the right of D_L. Let D_R

be the $\omega \times \omega$ matrix whose entry $1,1$ is 1, and whose other entries are 0. This matrix will select the first column when multiplied on the left of any matrix. Using additional matrices S_1 to S_m uniformly at random and invertible, m garbled matrix groups can be built:

$$\begin{aligned} U_1 &= D_L M_1 S_1 \\ U_i &= S_{i-1}^{-1} M_i S_i && \text{for } i \in \{n \in \mathbb{N} \big| 1 < n < m\} \\ U_m &= S_{m-1}^{-1} M_m D_R \end{aligned}$$

Hence, each $U_{1..m}$ does not reveal usable information by itself [CFIK03], but $\prod_{i=1}^{m} U_i$ does still calculate the desired result.

7.1.4 Evaluating the Matrices using OAFEs

From Section 7.1.3 the matrices $U_{1..m} \in K^{\omega \times \omega}$ are obtained. The matrices $U_{1..m}$ can be reshaped to vectors $u_{1..m} \in K^{\omega^2}$. The vectors $u_{1..m}$ can be concatenated to one large vector $\mu \in K^{m\omega^2}$. Next, two vectors a and b are deduced that hold the following property ($a, b \in K^{m\omega^2}$, x_D a scalar variable, as in Section 7.1.2 David's input):

$$a \cdot x_D + b = \mu \tag{7.1}$$

Using the $\prod_{\text{OAFE}}^{\text{semi-int}}$ protocol [DKMQ12] the function can now be evaluated securely.

- The setup is: a, b as above, $\mathbb{F}_q = K$, $k = m\omega^2$ and m known to both, David and Goliath
- After applying the protocol, David is now able to evaluate the linear functions to his result vector $y \in K^{m\omega^2} = GWh + \tilde{a}x_D + \tilde{b}$ (G, W, h, $\tilde{a}$,$\tilde{b}$ as in the paper [DKMQ12])
- The last but one step is to reshape y to the matrices $F_{1..m} \in K^{m\omega^2}$
- Finally, the entry $2,1$ of $\prod_{i=1}^{m} F_i$ is the desired result of $f(x_G, x_D)$.

7.2 Decomposable Affine Randomized Encodings

The second approach that was examined during this thesis, was to transform general functions to affine functions using a slightly modified form of the *Decomposable Affine Randomized Encodings* (DAREs) from *Garbled Arithmetic Circuits* by Applebaum et al. [AIK12]. The change is necessary because this thesis uses OAFEs to evaluate the DAREs and does therefore not depend on the *learning with errors* (LWE) problem. This leads to further changes: The *affinization gadget* [AIK12] is modified and the *key–shrinking gadget* [AIK12] is not needed. Although the final version of this thesis does not directly contain anything from Applebaum's garbled arithmetic circuits, some ideas are closely related and inspired by Applebaum's paper.

7.2.1 Definitions

These definitions are not from the original paper but are closely related. A *Linear Randomized Expression* (LRE) is an element of the set $\mathcal{F}_{AR}$ (K a finite field), a *Decomposable Affine Randomized Encoding* (DARE) an element of the set $\mathcal{E}_{AR}$. One important constraint has to hold for all LREs and therefore for all DAREs too: After fully evaluating an LRE, i.e. replacing the variable by its actual value, the now constant LRE should reveal no more information than the result of a full decoding of the respective DARE. The safety is proved in *How to Garble Arithmetic Circuits* [AIK12].

$$\begin{aligned}\mathcal{V} =& \{x \mid x \text{ a variable over } K\}\\ \mathcal{F}_{AR} =& \{s \cdot x + i \mid s, i \in K, x \in \mathcal{V}\} \cup \{v \mid v \in K\}\\ \mathcal{E}_{AR} =& \{(M, A) \mid M \subseteq \mathcal{F}_{AR} \times \mathcal{F}_{AR}, A \subseteq, \mathcal{F}_{AR}; A, M \text{ finite multi–sets}\}\end{aligned}$$

7.2.2 Encoding

- A function $f_1(x_1, x_2) = x_1 + x_2$ can be securely encoded by $ENC_A(f_1, r)$; $r \in K$ being uniformly at random.
- A function $f_2(x_1, x_2, x_3) = x_1 \cdot x_2 + x_3$ can be securely encoded by $ENC_M(f_2, r_1, r_2, r_3, r_4)$; $r_1, r_2, r_3, r_4 \in K$, uniformly at random.

$$\begin{aligned}ENC_A(f_1, r) &= \Big(\emptyset, (x_1 + r, 1 \cdot x_2 - r)\Big)\\ ENC_M(f_2, r_1, r_2, r_3, r_4) &= \left(\left\{ \begin{pmatrix} 1 \cdot x_1 - r_1 \\ 1 \cdot x_2 - r_2 \end{pmatrix} \right\}\right.\\ &\quad ,\Big\{ r_2 \cdot x_1 - r_1 r_2 + r_3\\ &\quad ,\ r_1 \cdot x_2 + r_4\\ &\quad ,\ 1 \cdot x_3 - r_3 - r_4 \Big\}\Big)\end{aligned}$$

7.2.3 Decoding

Decoding a fully evaluated DARE $\mathcal{E}_{AR} = \{(M, A)\}$ as $r = DEC(\mathcal{E}_{AR})$ is straightforward:

$$\begin{aligned}M' &= \left\{ m_1 \cdot m_2 \;\middle|\; \begin{pmatrix} m_1 \\ m_2 \end{pmatrix} \in M \right\}\\ r &= \sum_{a \in A} a + \sum_{m \in M'} m\end{aligned}$$

7.2.4 Randomized Variables

Whenever a circuit is not directly transformable to one single DARE, sub–DAREs get replaced by *Randomized Variables* (RV). RVs do not appear in Applebaum's paper [AIK12]. RVs are just (sub–)DAREs that get transmitted to the second party (see Section 7.2) which fully evaluates them and saves the result just like an ordinary input variable (such as x_D). Following DAREs may then use the RVs as pseudo–inputs. But since a DARE reveals as much information as its decoded form, the original DARE cannot just be transmitted as the final overall DARE. That would reveal intermediate information. Therefore, RVs get an additional garbling step: Say the following property holds for a variable v being the decoded value of a DARE e:

$$v = DEC(e \in \mathcal{E}_{AR})$$

Then a modified DARE $\hat{e} \in \mathcal{E}^{+}_{AR}$ decoding to a value $\hat{v}$

$$\hat{v} = \alpha \cdot (v + \beta) = DEC(\hat{e})$$

with secret keys α and β—known only by the first party—is transmitted.

7.2.5 Relation to the Final Version

Some of the ideas and entities presented in this section relate to the final version of this thesis. *Decomposable Affine Randomized Encodings* (DAREs) are closely related to *Dual Randomized Affine Encodings* (DRAEs, Section 2.4). The difference is that an evaluated DARE is equivalent to the value of the sub–circuit it models, DRAEs in contrast are still encrypted (twofold). Because the DAREs are equal to their plain evaluation, the old version presented in this section needed the *Randomized Variables* (RVs) that are not present in the final version. However, the final version also assigns evaluated sub–circuits to variables, but since DRAEs are encrypted anyways, there is no need for an additional encryption.

7.2.6 Implementation State

The ideas of this section were implemented, but the implementation did evolve to the final version described in Chapter 2. If there is interest in the old implementation, it can be recovered from this thesis' source code tree by using the git version control system.

Bibliography

[AIK12] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:58, 2012.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 136–145. IEEE, 2001.

[CFIK03] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. *Advances in Cryptology—EUROCRYPT 2003*, page 642, 2003.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. URL: `http://doi.acm.org/10.1145/351240.351266`, `doi:10.1145/351240.351266`.

[Cha00] Manuel M.T. Chakravarty. C $\longrightarrow$ haskell, or yet another interfacing tool. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, volume 1868 of *Lecture Notes in Computer Science*, pages 131–148. Springer Berlin Heidelberg, 2000. URL: `http://dx.doi.org/10.1007/10722298_8`, `doi:10.1007/10722298_8`.

[Cle91] R. Cleve. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1(1):91–105, 1991.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2001.

[CW79] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[DKMQ12] Nico Döttling, Daniel Kraschewski, and Jörn Müller-Quade. David & goliath oblivious affine function evaluation—asymptotically optimal building blocks for universally composable two-party computation from a single untrusted stateful tamper-proof hardware token. *IACR Cryptology ePrint Archive*, 2012:135, 2012.

[HMPJH05] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[Knu81] D.E. Knuth. *The Art of Computer Programming: Seminumerical algorithms*. The Art of Computer Programming. Addison-Wesley Pub. Co., 1981.

[Len11] Mark Lentczner. 32–bits less is more. *Zero Knowledge*, 2011. URL: `http://mtnviewmark.wordpress.com/2011/12/07/32-bits-less-is-more/`.

[M+] S. Marlow et al. Haskell 2010 language report. URL: `http://www.haskell.org/definition/haskell2010.pdf`.

[NP99] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 245–254. ACM, 1999.

[NP06] M. Naor and B. Pinkas. Oblivious polynomial evaluation. *SIAM Journal on Computing*, 35(5):1254–1281, 2006.

[NPS99] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

[Rab81] M.O. Rabin. How to exchange secrets by oblivious transfer. Technical report, Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[Yao82] A.C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[Yao86] A.C.C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.