FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A Qualitative Study of Indistinguishability Obfuscation

Nils Kunze

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A Qualitative Study of Indistinguishability Obfuscation

Eine qualitative Studie zu "indistinguishability obfuscation"

Author:	Nils Kunze
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	Dr. Martin Ochoa, Sebastian Banescu
Submission Date:	September 15, 2014

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, September 15, 2014 Nils Kunze

Acknowledgments

I want to thank my advisors Martin Ochoa and Sebastian Banescu for allowing me to approach this highly interesting topic. They gave me great guidance and pushed me when I did not work as much as I should have.

I also want to thank my friends for bearing with my complaints when I again piled up too much work and my flatmates for always cheering me up. You are the best! A special thanks goes out to Sophie for reviewing a very ugly first draft.

Abstract

Code obfuscation is a powerful tool to prevent reverse engineering which is beneficial for software authors and software users alike. Despite their importance, most obfuscation methods used in practice today do not have any formal security guarantees, making it difficult to trust in their resilience. These formally secure constructions would not only bring security benefits, but also have highly interesting applications. Examples of such applications include multi-party computation or devising new encryption schemes with desirable properties such as deniable or functional encryption.

Recently Garg et al. [1] proposed the first candidate construction for an obfuscator which accomplishes the notion of indistinguishability obfuscation. This thesis focuses on examining the memory and time overheads that are implied by the suggested construction, with the goal to understand its practical applicability. We implemented the described construction as closely as possible in Python, using the libraries of Sage [2] to deal with the more advanced algebraic objects. We then determined the runtime and memory costs of generating different steps in the construction for circuits of different sizes as well as their implied overhead during evaluation.

We find that we are not able to actually generate a complete obfuscation for any but the smallest circuits, because the growth in memory is too big and the generation too slow. We project that even for circuits with less than 10 gates the obfuscation process could take longer than 10^7 years. Since interesting circuits are usually orders of magnitudes bigger than that, we conclude that the overhead of this candidate is too high to have any meaningful practical applications just yet.

Zusammenfassung

Obfuskatoren schützen sowohl Software Autoren als auch Nutzer, indem sie Reverse Engineering verhindern oder erschweren. Trotz ihrer Wichtigkeit gibt es bis heute keine Obfuskatoren, deren Sicherheit formell abgesichert ist. Wenn es Obfuskatoren gäbe, deren Sicherheit formell bewiesen ist, hätten diese zahlreiche interessante Anwendungen, zum Beispiel im Bereich von Multi-Party Computation oder zum Entwickeln von neuen Verschlüsselungsmethoden.

Im letzten Jahr haben Garg et al. [1] eine Konstruktion vorgestellt, die den formellen Anforderungen von Indistinguishability Obfuscation entspricht. In dieser Arbeit untersuchen wir den Speicherverbrauch und den zeitlichen Mehraufwand dieser Konstruktion mit dem Ziel die praktische Anwendbarkeit zu ermitteln. Wir haben die beschriebene Konstruktion so nahe wie möglich in Python implementiert und dabei Sage [2] benutzt um kompliziertere algebraische Objekte zu behandeln. Im Anschluss haben wir die Laufzeit und den Speicherbedarf zum Generieren verschiedener Zwischenschritte der Konstruktion untersucht und analysiert wie sich die Laufzeit verhält.

Aufgrund des extrem hohen Speicherbedarfs und der sehr langsamen Generation, konnten wir die Konstruktion nur auf sehr kleine Schaltkreise anwenden. Unsere Hochrechnungen ergeben, dass das Verschleiern von Schaltkreisen mit weniger als 10 Gattern bereits länger als 10^7 Jahre dauern würde. Da praktisch relevante Schaltkreise aus wesentlich mehr Gatter bestehen, ist unsere Schlussfolgerung, dass diese Konstruktion in ihrer jetzigen Form nicht praktisch anwendbar ist.

Contents

1 Introduction

Whenever a software author would like to distribute his program, he has to make it available to his users in some form or another. Depending on the form he chooses, it can become susceptible to a variety of attacks. The most secure form would be to only make the software available for querying on a server. While it would in theory be possible, this is normally prevented by the performance impact, availability problems or other real world issues. Whenever a server-based solution is not possible, a distribution in binary form is the next secure one, because it is difficult for a human to extract any information from a computer-executable binary. Nonetheless, it is possible that a malicious attacker might try to analyze the binary in order to gain some additional power or knowledge, a process which is known as reverse engineering.

There are numerous reasons as to why an attacker would have an interest in reverse engineering a given program. As an example imagine a shareware program that is limited in functionality or can only be used for a limited amount of time. In order to be able to use the program without paying, a user might use reverse engineering in an attempt to remove or trick the mechanism that is enforcing these restrictions. Another scenario is that the program contains some kind of new technology (e.g. a new algorithm or data structure) so that a competitor of the releasing party might try to gain a competitive advantage by understanding the code and extracting the new technology. While there are legal defenses that can prevent the theft of intellectual property, they are sometimes not enough, because infringement can be exceedingly difficult to detect and pursue (legal costs). Finally, criminals trying to attack other users of the same program often use reverse engineering to find bugs in the code and to understand how to best exploit them.

1.1 What is Obfuscation?

The process of trying to impede reverse engineering is called obfuscation. Code obfuscation tries to maintain the functionality of the code, while at the same time making it harder to understand the program. As an example, in Figure 1.1 you can see a heavily obfuscated piece of C code, which stems from the IOCCC (The International Obfuscated C Code Contest). Understanding the behavior of this code snippet is vastly more difficult than understanding 'normal' C code, because it removed nearly all whitespace, reduced variable names and generally uses a lot of clever tricks aimed at making the understanding harder. Even so, the code is still doing what it was originally developed to do, in this case it is an ASCII/Morse code translator. While this kind of obfuscation very likely has no impact on the difficulty of reverse engineering the compiled binary, it is quite useful to illustrate the general idea of obfuscation: It tries to make it hard to understand the functionality without interfering with it.

```
#include<stdio.h> #include<string.h> main(){char*O,l[999]=
"’‘acgo\177~|xp .-\0R^8)NJ6%K4O+A2M(*0ID57$3G1FBL";while(O=
fgets(l+45,954,stdin)){*l=O[strlen(O)[O-1]=0,strspn(O,l+11)];
while(*O)switch((*l&&isalnum(*O))-!*l){case-1:{char*I=(O+=
strspn(O,l+12)+1)-2,O=34;while(*I&3&&(O=(O-16<<1)+*I---’-’)<80);
putchar(O&93?*I&8||!( I=memchr( l , O , 44 ) ) ?’?’:I-l+47:32);
break;case 1: ;}*l=(*O&31)[l-15+(*O>61)*32];while(putchar(45+*l%2),
(*l=*l+32>>1)>35);case 0:putchar((++O,32));}putchar(10);}}
```

Figure 1.1: ASCII/Morse code translator by Frans van Dorsselaer, winner of the 1998 IOCCC, taken from [3]

Code obfuscation is used to prevent the exploitation of programs by making it harder to find and exploit bugs. Nowadays using it is often a practical necessity, because languages like Java can be easily decompile if it not used. This can cause big problems, not only for the users of the software, but also for the authors, when their non-free programs are cracked and freely distributed over the Internet.

1.2 State of the Art

In order to obfuscate a program, one needs to transform the code in a way that does not change its functionality, but makes reverse engineering of the program harder. There are a number of techniques which are frequently used nowadays and they can generally be classified into one of the following classes[4]:

Control flow obfuscation The general idea is to modify the programs control flow in a way that thwarts analysis. One technique that falls into this category is control flow graph flattening [5], which effectively raises the number of possible paths in a program, thus making static analysis much harder.

Another option is the insertion of opaque predicates [6] at arbitrary positions in the code. Opaque predicates always evaluate to the same value, but it is hard to calculate that value statically. Thus it is possible to add a lot of conditional jumps to the program, whose behavior is difficult to understand statically, while always taking the same path during runtime.

Data obfuscation In order to make it more difficult for an attacker to understand where in the program a specific value is used in what way, it is possible to apply a number of data transformations. For example it is possible to change the encoding, to shuffle data that was ordered originally, or to merge data that doesn't belong together. Finally, it is possible to encrypt the data that is stored in memory and to only decrypt it, when it is accessed. Of course the key for decryption has to be available to the program in some way, which means that it is possible for an attacker to recover it, but static analysis will be much harder and one can try to hide the key in clever ways.

Code encryption Corresponding to encrypting data, it is also possible to encrypt the code of the program [7]. Just as with data encryption, there is a decryption routine with an embedded key in the clear which will then decrypt the rest of the program on the fly as needed. Again, this does help against static analysis, but during execution the actual code will be available in the clear, thus it is not too difficult to break.

There are a lot of different solutions [8][9][10][11] which make use of techniques which fall into one ore more of these categories, but all of them can be broken with dynamic, sometimes even with static attacks. This is not catastrophic, in the sense that these techniques fail to accomplish what they set out to do, namely making the progress of reverse engineering more difficult. It does point to the underlying problem though, which is, that these methods cannot offer meaningful formal security guarantees, so that we cannot have any confidence in their resilience. We can hope that they do slow down an attacker a bit, but it is difficult to quantify by how much if at all.

1.3 Formally Secure Obfuscation

In order to have reasonable confidence in the security of an obfuscator, we would like it to have a rigorous definition of security, which offers us a relation to mathematical hardness assumptions. The goal is to have a constructions for which we know, that if it can be broken, then we found an algorithm for a problem that is (currently) known to be difficult (e.g. efficiently factoring hard integers [12][13]). In order to reach such a construction, first we need a clear mathematical definition about what it even means for an obfuscator to be secure.

1.3.1 Black Box Obfuscation

The notion of black box obfuscation captures the most natural definition of obfuscation. It requires that an obfuscator makes a program ’unintelligible’, while at the same time preserving its functionality. Here ’unintelligible’ means, that everything that can be computed given the obfuscated program, should also be efficiently computable only given oracle access - that is only input-output pairs. This is the so called "virtual black box" property and it essentially requires that the obfuscation is so good, that having an obfuscated version of the program is no better to an attacker than if there was a server that he can query on inputs on his choice. Additionally the obfuscator should only add a polynomial slowdown factor to the execution of the program.

Quite intuitively this kind of obfuscation is only interesting for non-learnable programs. A learnable program is a program whose complete source code (or behavior) can be determined by looking at a finite set of input-output pairs. A simple example of a learnable program would be a program that prints out its own source code (also known as ’quine’[14]). Even an obfuscator that does not change the program he gets as an input at all would achieve the virtual black box property for all learnable programs, because just having oracle access would be enough for any attacker to recover the source code, making it unnecessary to hide it at all.

If we could construct an obfuscator with the capabilities described above, there would be a lot of interesting applications. Obviously it would yield strong software protection against reverse engineering as discussed above, but there are several other highly interesting applications in cryptography as well. For example would it be possible to turn any public-key cryptosystem into a homomorphic encryption scheme. To do so, one would simply need to construct a program that first decrypts the inputs it receives with an embedded key, then applies the required operations and finally encrypts the result again with the same key. Then just take this program and obfuscate it. Since the obfuscation makes it impossible to recover the embedded key, this would create a secure homomorphic encryption scheme. The same trick can be used to turn a private-key encryption scheme into a public-key one. Simply create a program that uses an embedded key to encrypt its inputs using the private-key encryption scheme and obfuscate it. Everybody will be able to encrypt messages using the obfuscated program, but only the owner of the private key will be able to decrypt messages, since it is not possible to recover the key.

Unfortunately Barak, et al. showed [3] that black box obfuscation is not possible in the general sense. In fact, they show by construction, that there exists a family of functions that is inherently unobfuscatable. For any function from this family, there exists some property which can always be computed if you have access to a program that computes the function, but it cannot be determined by just looking at the input-output pairs. The result is strong, because every obfuscator fails completely on these programs. It is not even possible to hide a single bit of information about these programs, which effectively means that one could recover the complete source code. Additionally these programs are strongly non-learnable, which in a way makes them close to the programs one would like to obfuscate in practice. On the other hand, the result is weak, because obfuscation only fails on some programs. In fact there might be a large class of programs that are indeed black box obfuscatable.

1.3.2 Indistinguishability Obfuscation

All in all the impossibility result presented in the previous sections shows that in order to gain a meaningful definition of secure obfuscation, it is necessary to change something. Basically there are two options: either restrict the class of functions that need to be obfuscated, so that the restricted class excludes the functions that were shown to be unobfuscatable, or relax the security requirements of the obfuscator. It is not exactly clear what other functions might be unobfuscatable in the black box sense, thus it makes more sense to strive for a definition of obfuscation that can be achieved for general programs. This is where the notion of indistinguishability obfuscation comes into play, which only requires that given two similar sized circuits which implement the same function, their obfuscations should be computationally indistinguishable. To clarify, given two programs P_0 and P_1 which are functionally equivalent, i.e. $P_0(x) = P_1(x)$ for all x, if one applies an indistinguishability obfuscator to both of them, it should not be possible to determine which obfuscated program stems from which original one.

It is unclear, what kind of security this definition can actually offer in practice. On the other hand it has been shown[15] that indistinguishability obfuscation achieves the notion of best-possible obfuscation, offering at least 'a strong philosophical justification'[1]. Best possible obfuscation requires that the obfuscated program leaks as little information as any other program that implements the same functionality and is of similar size. This means that it is indeed the best possible obfuscation for any program that implements this functionality.

One practical use of indistinguishability obfuscation is the removal of software watermarks. When handing out software to different people, it is possible to mark each copy uniquely without interfering with the functionality of the software, for example by changing the order of independent operations. This kind of watermark can then for example be used to understand who uploaded the program to a file-sharing site. Since watermarks are usually in the code, but do not have any impact on the functionality of the program, simply applying an indistinguishability obfuscator to it, will remove the software watermarks.

Apart from this, the practical uses of indistinguishability obfuscation are somewhat unclear, but there are a lot of theoretical cryptographic constructions that can be built from them, so many in fact that some go as far as envisioning it as a ’central hub for cryptography’ [16]. In fact it is at least possible to construct the following: Functional encryption [1], deniable encryption, public key encryption, non-interactive zero knowledge proofs and oblivious transfer (all presented in [16]).

1.4 Thesis Intention

Quite recently there was a breakthrough in the study of obfuscation, when Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai and Brent Waters presented a candidate construction for indistinguishability obfuscation for all circuits. Until then it was actually not know, how to construct such an obfuscator. Especially interesting is that this construction might even be a black box obfuscator for a large class of circuits - namely all that are not excluded by the impossibility result. While it is encouraging to see such a big step forward in the area of obfuscation, we were interested in the question of practicality of this construction. Even the authors themselves say:

> "While our current obfuscation construction runs in polynomial-time, it is likely too inefficient for most practical problems."

The goal of this thesis is to understand what performance overhead one would have to expect when working with this construction. Even if it is indeed too slow to obfuscate a whole program, there might be applications where the tradeoff between performance and security is indeed worth it, especially when one only obfuscates small parts of the whole program.

2 Preliminaries

There are a number of mathematical constructions and principles that will be used in the construction of the candidate obfuscator for $\mathbf{NC}^1$ circuits. They will be presented in this chapter.

When $x \in \{0,1\}^n$ is a bitstring of length n, x_i will mean the value of the bit at position i. PPT is a shorthand for probabilistic polynomial-time Turing machine. A function $a(n)$ is called negligible, if it grows slower than any inverse polynomial. That is for every positive polynomial $p(n)$, there exists some n_0 so that for all $n > n_0$ it holds that $f(n) < \frac{1}{p(n)}$. When M is a Turing machine and f a function, we write $M^f(\cdot)$ to mean the execution of M with oracle access to f.

2.1 Circuits

Circuits are a model of computation that is commonly used in cryptographic constructions, because they are reasonably powerful while still being relatively simple. Their power stems from their universality which means that it is possible to express any function using circuits. On the other hand, their simplicity enables us to focus on the core of the problem while leaving aside inessential details.

Definition 1 (Circuit)**.** A circuit is a labeled directed acyclic graph. The vertices of the graph with in-degree > 0 are called *gates* and are labeled with $\{\neg, \wedge, \vee\}$, while those vertices with in-degree 0 are called *inputs* to the circuit and labeled with $x_1, ..., x_l$. Gates labeled as $\neg$ must have an in-degree of 1 and all others must have an in-degree of 2. We call those gates with out-degree 0 the outputs of the circuit. The *size* of a circuit is the total number of gates, while the *depth* of a circuit is the length of the longest path from any input to any output gate, that exists in the circuit.

To evaluate a circuit C one assigns values from $\{0,1\}$ to all of the inputs of the circuit and then recursively evaluates each of its gates, that is each gates computes the function that it is labeled with over its inputs. The value of the complete circuit is then defined as the value of its output gates.

For ease of notation we write $C(a)$ with $a \in \{0,1\}^l$ to mean that input x_i will be assigned value a_i, before the circuit will be evaluated. It is possible to understand a circuit with l inputs and k outputs as computing a function from $\{0,1\}^l$ to $\{0,1\}^k$. Usually the input size for a function is not fixed in advanced, thus in order to describe a general function from $\{0,1\}^*$, it is customary to consider circuit families. A circuit family is a set of circuits $\{C_i\}$ with $i \in \mathbb{N}$, where each circuit C_i computes the function for i inputs.

2.1.1 Nick's Class (NC)

NC is a circuit complexity class named after Nick Pippenger ('Nick's Class'), who first defined the class and did a considerable amount of research on it [17]. NC contains the circuits with l inputs and depth in $O(\log^c l)$ where $c \in \mathbb{N}$. An equivalent definition of NC describes it as the set of decision problems which are efficiently solvable by a parallel computer. This requires that a computer with a polynomial number of processors must be able to solve the problems in polylogarithmic time. It is known that $NC \subseteq P$, because it is possible to simulate parallel computers with sequential ones. On the other hand it is unknown whether $NC = P$, but it is thought to be improbable because there are probably problems in P that cannot be sped up by using parallelism.

It is useful to further categorize NC into subsets NC^i, where each NC^i corresponds to the set of decision problems solvable by circuits with l inputs, a polynomial number of gates (where each gate has a maximum of two inputs) and of depth $O(\log^i l)$. It does then hold that $NC = \bigcup_{i \in \mathbb{N}} NC^i$.

2.2 Notions of Obfuscation

This section contains formalizations of the notions of obfuscation that were mentioned in the introduction.

Definition 2 (Black box obfuscator)**.** A PPT $\mathcal{O}$ is a black box obfuscator for a circuit family $\mathcal{C}_\lambda$ if it satisfies the following conditions:

- (Functionality preserving) For all security parameters $\lambda \in \mathbb{N}$, all circuits $C \in \mathcal{C}_\lambda$ and all inputs x it holds that

$$Pr[C'(x) = C(x) : C' \leftarrow \mathcal{O}(\lambda, C)] = 1$$

- (Polynomial slowdown) For all security parameters $\lambda \in \mathbb{N}$ and all circuits $C \in \mathcal{C}_\lambda$, $|\mathcal{O}(\lambda, C)| \leq p(|C|)$ where p is some polynomial.

- ("virtual black box") For every PPT A there exists another PPT S and a negligible function a, so that:

$$|Pr[A(\mathcal{O}(\lambda, C)) = 1] - Pr[S^C(1^C) = 1]| \leq a(|C|)$$

The virtual black box property requires that for every predicate that the adversary A can compute from the obfuscation of the circuit, there exists a simulator S which can compute the same predicate only using oracle access. While the adversary A could actually compute a variety of things, e.g. an output distribution, a relation or a function over the obfuscated program, it is easiest to compute a predicate, i.e. a function with image $\{0,1\}$, which is the same as an arbitrary property of the program.

While this definition of Black Box Obfuscation is already too weak for a lot of the practical applications discussed in section 1.3.1, even this weak definition is impossible in the general sense. There is a proof presented in [3] which shows by construction that there exists a family of functions $\mathcal{F}_k$ that is inherently unobfuscatable in the black box sense. This means that there exists a predicate $\pi : \mathcal{F} \to \{0,1\}$ such that for any program that computes a function $f \in \mathcal{F}_k$, $\pi(f)$ can be computed efficiently. On the other hand, no efficient algorithm can compute $\pi(f)$ with probability much better than guessing if it only has oracle access. Formally for any PPT S, $Pr[S^f(1^k) = \pi(f)] \leq \frac{1}{2} + a(k)$ where a is some negligible function and $f \in \mathcal{F}_k$. On the other hand there exists another PPT A so that for any $f \in \mathcal{F}_k$ and any circuit C_f which computes f it holds: $A(C_f) = \pi(f)$. The construction of the family $\mathcal{F}_k$ relies on the existence of one-way function, but if black box obfuscators exist, one-way function also exist, which causes a contradiction. For the full construction we refer to [3].

The impossibility of Black Box Obfuscation brings us to the notion of Indistinguishability Obfuscation, which is the one that is actually relevant for this thesis.

Definition 3 (Indistinguishability Obfuscator)**.** A PPT $i\mathcal{O}$ is called an indistinguishability obfuscator for a family of circuits $\mathcal{C}_\lambda$, if it meets the functionality preserving and polynomial slowdown properties described above and additionally has the following property:

- (indistinguishability obfuscation) For every PPT distinguisher D, all security parameters $\lambda \in \mathbb{N}$ and all circuits $C_0, C_1 \in \mathcal{C}_\lambda$ where $C_0(x) = C_1(x)$ for all x, there exists a negligible function α so that

$$|Pr[D(i\mathcal{O}(\lambda, C_0)) = 1] - Pr[D(i\mathcal{O}(\lambda, C_1)) = 1]| \leq \alpha(\lambda)$$

For the understanding of this requirement it can be helpful to recall the distinguisher game which is often used to describe properties of cryptographic constructs. Here the adversary tries to distinguish the obfuscations of two programs. He hands two programs C_0, C_1 which both compute the same function to the obfuscator, who then applies the indistinguishability obfuscation to them and returns only one. The adversary then needs to decide if he received an obfuscation of C_0 or C_1. Essentially the last property requires that the adversary cannot tell which program he got in return with probability much better than guessing.

As mentioned in chapter 1.3.2, any indistinguishability obfuscator also attains another notion of obfuscation, namely best possible obfuscation.

Definition 4 (Best-possible obfuscator)**.** A PPT $b\mathcal{O}$ is called best-possible obfuscator for a family of circuits $\mathcal{C}_\lambda$, if it preserves the functionality and only implies a polynomial slowdown as described above and also has the following property:

- (best-possible obfuscation) For every PPT learner L there exists a PPT simulator S, so that for any two circuits $C_0, C_1 \in C_\lambda$ where $C_0(x) = C_1(x)$ for all inputs x and $|C_0| = |C_1|$ it holds that the two output distributions $L(b\mathcal{O}(\lambda, C_0))$ and $S(C_1)$ are indistinguishable.

To paraphrase the last property: Whatever information one can extract efficiently from the obfuscation of C_0, one can also learn (simulated) from every other circuit that has the same size and functionality. Thus whatever information the best-possible obfuscator cannot hide, can be extracted from any circuit that implements the same functionality, making it in fact the best obfuscation that is possible.

To show that every efficient indistinguishability obfuscator is also an efficient best-possible obfuscator consider the following: Let $i\mathcal{O}$ be an efficient indistinguishability obfuscator. Then for every learner L, let S be the simulator that receives C_1 as an input, generates $i\mathcal{O}(C_1)$ and then starts $L(i\mathcal{O}(C_1))$. Since $C_0(x) = C_1(x)$ for all inputs x, it follows that the output of $L(i\mathcal{O}(C_0))$ and $S(C_1) = L(i\mathcal{O}(C_1))$ must be indistinguishable, because $i\mathcal{O}(C_0)$ is indistinguishable form $i\mathcal{O}(C_1)$. If the learner was able to extract some more information from one of them, it would be possible to distinguish them using that information and $i\mathcal{O}$ would not be an indistinguishability obfuscator - a contradiction. For a more in depth treatment of this property of indistinguishability obfuscators, we refer to [15].

2.3 Branching Programs

Branching Programs are yet another model of computation that will be used in this paper. In order to understand their powers and limitations, it is useful to start with the notion of binary decision diagrams, which can be seen as a compressed way to express boolean functions.

Definition 5 (Binary Decision Diagram)**.** A binary decision diagram is a rooted, directed, acyclic graph, with all but two nodes being labeled with one of the input variables. The last two nodes are called terminal nodes and have an out-degree of 0. The first one is labeled as 'true' and the other one as 'false'. Every node besides the terminal nodes have an out-degree of 2 and these two outputs are labeled true or false respectively.

In order to evaluate a binary decision diagram on a particular assignment of the input variables, we follow the path from the root according to the values of the input bits. More exactly, for each node that we come across, we look at the value of the input variable corresponding to the label of the node. If it is true, we take the edge labeled as true, otherwise the one labeled as false. The label of the terminal node that is reached when following this procedure, determines the result of the evaluation of the binary decision diagram on that particular input.

We define one *layer* of a binary decision diagram by those nodes with the same distance from the root node. The width of a binary decision diagram is the maximum number of nodes in any layer.

To present the transformation of Binary Decision Diagrams to Branching Programs, it is useful to only look at binary decision diagrams that conform with the following restrictions:

- all layers have the same number of nodes
- all nodes of one layer are labeled with the same variable
- between two adjacent layers the edges labeled as true form a permutation, as do the ones labeled as false

This brings us to the definition of branching programs:

Definition 6 (Branching Program)**.** A Branching Program is a sequence of instructions of the form $< inp(i), \sigma, \tau >$, where $inp(i) \in [l]$ tells us which bit to look at in step i and σ and τ are permutations. Additionally the Branching Program contains two permutations A_0 and A_1. To evaluate a branching program on a binary input $X = x_1x_2...x_l$ with each $x_i \in \{0,1\}$, we first evaluate each instruction by looking at input bit $x_{inp(i)}$ and choosing σ if it is 0 and τ otherwise. This yields a sequence of permutations, that can be reduced by chaining them all together. If the resulting permutation is equal to A_0 the Branching Program evaluates to 0, if it is equal to A_1 it evaluates to 1.

We call the number of instructions in a Branching Programs its *length n*. The length of each of the permutations is the *width* of the Branching Program.

It is obvious that the definition of Branching Programs is identical to the restricted Binary Decision Diagram as described above. The permutation that is formed by the edges labeled as false are the first permutation in an instruction, while permutation formed by the true edges is the second one. The variable that the nodes in that layer look at is the value of $inp(i)$.

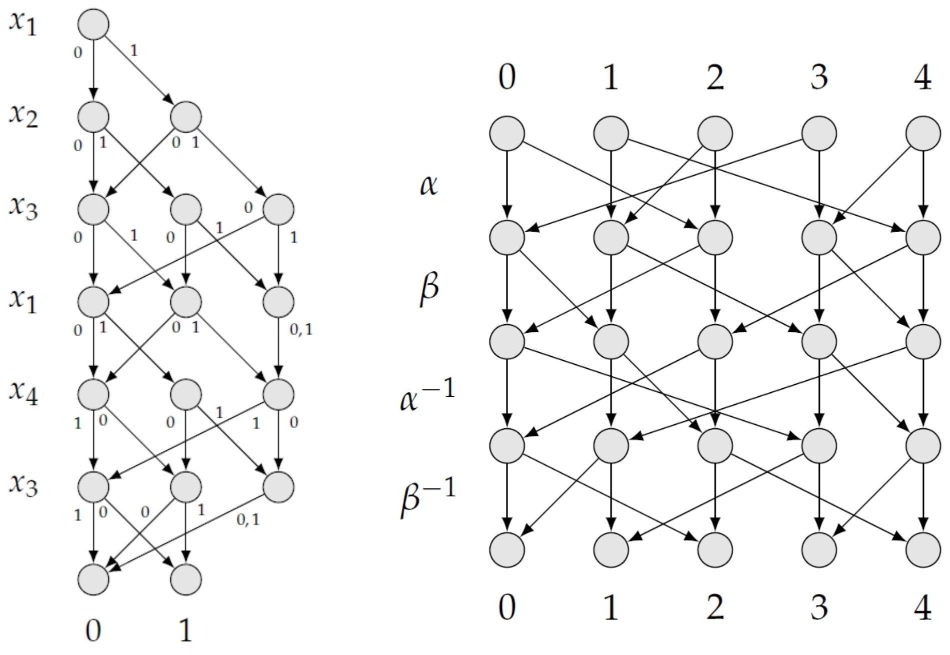

Figure 2.1: A Binary Decision Diagram ordered by layers and an example for permutations between layers (source: [18])

2.3.1 Barrington's Theorem

Barrington's Theorem proves that branching programs of bounded width and polynomial size are exactly $\mathbf{NC}^1$. This result was surprising, when it was first published, because it showed that computation in constant space is more powerful than expected. Especially when looking at the majority function, it is intuitive why Barrington's result was surprising: One would expect that to compute the majority we need an accumulator with $O(\log l)$ bits, but instead passing $O(1)$ bits is enough.

The following proof closely follows the one presented in [19]

Theorem 1 (Barrington's Theorem). *For any fan-in-2 Boolean circuit C with depth d, there exists a branching program of constant width and length smaller than 4^d that computes the same function.*

Definition 7 (5-cycle recognition). A program P in this proof is a slightly simplified version of branching program of width 5 which do not output 0 or 1, but instead the product of all their evaluated instructions $P(x) = \prod_{i=1}^{n} A_{i,x_{inp(i)}}$. We say that P σ recognizes a language $L \subseteq \{0,1\}^n$, if there exists a 5-cycle σ so that

- for all $x \in L : P(x) = \sigma$
- for all $x \notin L : P(x) = e$ (e is the identity permutation).

Lemma 1. *There exist 5-cycles σ, τ so that $\sigma\tau\sigma^{-1}\tau^{-1}$ is also a 5-cycle.*

Proof. $\sigma = (1\ 2\ 3\ 4\ 5), \tau = (1\ 3\ 5\ 4\ 2)$
$\sigma\tau\sigma^{-1}\tau^{-1} = (1\ 2\ 3\ 4\ 5)(1\ 3\ 5\ 4\ 2)(5\ 4\ 3\ 2\ 1)(2\ 4\ 5\ 3\ 1) = (1\ 3\ 2\ 5\ 4)$ □

Lemma 2. *For any two 5-cycles σ, τ, if there exists a program P that σ-recognizes a language L, there exists a program P' that τ-recognizes L.*

Proof. Because σ and τ are both 5-cycles, there exists a permutation φ so that $\tau = \varphi\sigma\varphi^{-1}$. The permutations of P' can be constructed like so $A'_{i,b} = \varphi A_{i,b}\varphi^{-1}$. When evaluating $P'(x) = \prod_{i=1}^{n} A'_{i,x_{inp(i)}} = \prod_{i=1}^{n} \varphi A_{i,x_{inp(i)}}\varphi^{-1} = \varphi P(x)\varphi^{-1}$, so that when $P(x) = \sigma$, $P'(x) = \varphi\sigma\varphi^{-1} = \tau$ and when $P(x) = e$, $P'(x) = \varphi e\varphi^{-1} = e$. □

Proof for Barrington's theorem. Given a circuit C of depth d that consists only of AND and NOT Gates, one can generate a program that recognizes the same language as the circuit with the following recursive procedure:

- Base case, d=0: C is either a variable or a constant. If $C(x) = x_i$ return the following program: $< i, e, \sigma >$, if $C(x) = 0$ return $< 1, e, e >$, if $C(x) = 1$ return $< 0, \sigma, \sigma >$.
- $C = \neg C'$.
  By induction there exists a P' that σ-recognizes C'. Generate P by copying all instructions of P', i.e. $A_{i,b} = A'_{i,b}$ with $i \in [n], b \in \{0, 1\}$. Subsequently modify both permutations in the last instruction of P like so: $A_{n,b} = A'_{n,b}\sigma^{-1}$ for $b \in \{0, 1\}$. Now P σ^{-1}-recognizes C.
- $C = C_0 \wedge C_1$.
  Once more by induction there exist programs P_0, P_1 so that P_0 σ-recognizes C_0 and P_1 τ-recognizes C_1. Additionally by Lemma 2 there exist programs P'_0, P'_1 so that P'_0 σ^{-1}-recognizes C_0 and P'_1 τ^{-1}-recognizes C_1. When σ and τ are chosen according to Lemma 1, the concatenation of these programs $P = P_0P_1P'_0P'_1$ 5-cycle recognizes C as demonstrated by this truth table:

$C_0(x)$	$C_1(x)$	P_0	P_1	P'_0	P'_1	P
0	0	e	e	e	e	e
1	0	σ	e	σ^{-1}	e	e
0	1	e	τ	e	τ^{-1}	e
1	1	σ	τ	σ^{-1}	τ^{-1}	$(1\ 3\ 2\ 5\ 4)$

□

Observe that it is trivial to create a branching program from a program P which σ-recognizes some language, by simply copying the instructions and setting $A_0 = e, A_1 = \sigma$.

2.4 Kilian's Protocol

Kilian [20] presented a protocol, which allows two parties to jointly evaluate an $\mathbf{NC}^1$ circuit, without either party learning the others input. Furthermore only one party gets the result of the evaluation.

The protocol works as follows: Alice and Bob would like to evaluate the $\mathbf{NC}^1$ circuit C on inputs x and y, where x is Alice's part of the input and y Bob's. Neither party should learn anything about the others input and only Bob should learn the output of the circuit $C(x, y)$. Alice initiates the protocol by transforming C into a branching program of length n using Barrington's Theorem 1. She performs the transformation in a way such that A_1 is equal to the identity matrix. The permutations in each instruction shall be represented by 5×5 permutation matrices $A_{i,b}$ with $b \in \{0, 1\}$, so that each instruction has the form $< k, A_{i,0}, A_{i,1} >$ for $i \in [n]$. Next Alice chooses n random invertible matrices R_i over $\mathbb{Z}_p$ and uses them to create a randomized branching program by setting $\tilde{A}_{i,b} = R_{i-1} \cdot A_{i,b} \cdot R_i^{-1}$ for all $i \in [n]$ and $R_0 = R_n$. She then proceeds to send the $\tilde{A}_{i,b}$ to Bob.

Let $\chi = (x|y)$ be the concatenation of the inputs from Alice and Bob. Alice can then already choose the correct matrices for all instructions that look at her part of the input $\{A_{i,\chi_{inp(i)}} : i \in [n], inp(i) \leq |x|\}$ and send them to Bob. In the next step they use oblivious transfer in order to let Bob learn the matrices $\{A_{i,\chi_{inp(i)}} : i \in [n], inp(i) > |x|\}$ that correspond to his input. Oblivious transfer enables Bob to learn exactly one of the possible matrices $A_{i,0}$ and $A_{i,1}$ for each i, without Alice learning which one he chose.

Now Bob has all matrices that he needs and he can put them in order to compute the product $P = \prod_{i \in [n]} \tilde{A}_{i,\chi_{inp(i)}}$. Observe that all of the R_i besides R_0 and R_n are going to cancel out, so that P will be equal to $R_n \cdot A_r \cdot R_n^{-1}$ for $r = C(x, y)$. Since A_0 is the identity matrix P will be the identity, iff $C(x, y) = 1$. The oblivious transfer protocol prevents Alice from learning anything about Bobs input and Killian showed that Bob cannot learn anything about Alice input (besides what he can deduce from knowing y and $C(x, y)$), because of the randomization. While this construction only enables Alice and Bob to compute a one bit output, it is easily possible to repeat the protocol to get longer outputs.

2.5 Universal Circuits

The general idea of universal circuits is to have circuits which can take an input that consists of two parts. The first part corresponds to a binary description C' of another circuit C, while the second part is the input x on which we would like to evaluate that other circuit. The universal circuit U should then always calculate the output of C on the input, that is $U(C', x) = C(x)$ for all x.

This is of course a slight simplification. We have already established that circuits always have predefined input sizes, thus it is not possible to have one universal circuit for all input circuits. Instead we define the circuit family of universal circuits, which consists of all possible universal circuits $U_{s,l}$. For each $U_{s,l}$ it holds that for every circuit C of size s and with input size l, $U_{s,l}(C', x)$ for C' some binary representation of C and all x.

2.6 Multilinear Jigsaw Puzzles

As introduction to Multilinear Jigsaw Puzzles, first an explanation of how they can be understood when viewed in group terms. One central part of the puzzle is a multilinear map [21] over groups of prime order p:

$$e : G_1 \times G_2 \times \cdots \times G_k \to G_T$$

This kind of map takes k elements from their respective groups and maps them into the target group G_T. For each of the k elements it holds that the mapping is linear, when all but one of them is fixed. A valid *Multilinear Form* is anything that can be computed using group operations and the multilinear map. For example for $k = 3$ and $x_i \in G_1, y_i \in G_2, z_i \in G_3, w_i \in G_T$ a valid form would be something like $w_1 \cdot e(x_3^2, y_2 y_4, z_1^7) \cdot w_2^4 \cdot e(x_1 x_2^3, y_1^4 y_3, z_2)$. When one understands a given set of group elements as puzzle pieces, a Multilinear Form is a try to solve that puzzle. A puzzle is solved, if the Multilinear Form on these elements results in the unit of the target group $g_T^0 \in G_T$. Here the parallel to a normal puzzle is that there are only few ways in which the pieces fit together in order to solve the puzzle. A hardness assumption resembling the decisional Diffie-Hellman assumption [22] for $k = 3$ could be formulated like so: It is infeasible to distinguish between $(\{g_1, g_1^a\}, \{g_2, g_2^b, g_2^c\}, \{g_3, g_3^d, g_3^e\}, \{g_T, g_T^{abcde}\})$ and $(\{g_1, g_1^a\}, \{g_2, g_2^b, g_2^c\}, \{g_3, g_3^d, g_3^e\}, \{g_T, g_T^z\})$ where g_i are generators of G_i, g_T is a generator of G_T and it holds that $e(g_1, g_2, g_3) = g_T$.

2.6.1 Formalized

There are two recent proposals for implementing approximations of multilinear maps. This thesis uses one that is based on lattices [23], but it could in theory also be implemented using another one [24], which works over the integers. The construction is a graded encoding scheme which offers slightly different functionality than just a multilinear map. First off, the encoding of elements is randomized, which means that for one element there exist many valid encodings. Additionally it is possible to only partially evaluate a multilinear map to get elements in some intermediate group. When talking in group terms this would mean that when one multiplies elements from G_1 with elements in G_2 the result would be in some intermediate group $G_{\{1,2\}}$.

More formally, one can encode plaintext elements at different levels which are subsets of the index set $[k]$. Then the only operations that are possible are addition of elements that are encoded at the same level or multiplication of elements whose levels are disjunct. Say a, b are both encoded at some level $S \subseteq [k]$, then it is possible to add them to obtain an encoding of $a + b$. On the other hand given c encoded at level S_c and d encoded at level S_d with $S_c, S_d \subseteq [k]$ and $S_c \cap S_d = \emptyset$ one can multiply them and the result will be encoded at level $S_c \cup S_d$. Since the encoding process is randomized it is difficult to check if two strings encode the same plaintext element, in fact in the setting that we use it is only possible to check if an element represents an encoding of 0 at the highest level $[k]$.

Essentially a Multilinear Jigsaw Puzzle consists of two algorithms, a Jigsaw Generator and a Jigsaw Verifier. the Jigsaw Generator takes as input a security parameter λ and the multilinearity parameter k and the plaintext elements and outputs the puzzle pieces which are the encoded plaintext elements. These encoded elements can be understood as puzzle pieces, because it is only possible to combine them using group operations and the multilinear map. The Jigsaw Verifier takes those encoded elements and a Multilinear Form for combining them and tests if the Form solves the puzzle, that is if it evaluates to the appropriate encoding of 0. For the complete formalized description of Multilinear Jigsaw Puzzles we refer to Section 2.2 in [1].

3 Candidate Construction for $\mathbf{NC}^1$ Circuits

The construction presented by Garg et al. essentially consists of two parts. The first part is an indistinguishability obfuscator only for $\mathbf{NC}^1$ circuits, which is then used in the second part in an amplification effort to create an obfuscator for polynomial sized circuits.

This chapter presents the entirety of the $\mathbf{NC}^1$ candidate, a construction which starts with the modification of the oblivious evaluation of $\mathbf{NC}^1$ circuits that Killian presented. Unfortunately the simple construction has some flaws which can lead to attacks on the obfuscation. These problems will subsequently be explained and their solutions will be presented, before the entire candidate shall be illustrated. The last section deals with some of the limitations of this construction and gives a quick overview of the candidate for polynomial sized circuits alongside with the problems that prevent it from being useful in a practical setting right now.

3.1 Underlying Idea

Let us assume a scenario in which Alice has an $\mathbf{NC}^1$ circuit C that she would like to obfuscate and then send to Bob. The general idea of this candidate is to use Kilian's protocol, as described in 2.4, using a universal circuit U as the circuit that Alice and Bob would like to evaluate on their joint input. + relate C′ to smth in the uc part of the text (how to get input...) + Alice's part of the input will be a binary description C' of the circuit C that she would like to obfuscate, while Bob's part of the input is any input x that he would like to evaluate C on, making their joint input $\chi = (C'|x)$. In the original protocol Alice uses oblivious transfer, in order to make sure that Bob can only evaluate the circuit for exactly his input and not additional ones. In the case at hand this lets Bob learn $U(\chi) = U(C', x) = C(x)$ for one input x of his choice, but nothing else.

Giving Bob the ability to evaluate only one input of his choice cannot be understood as a valid obfuscation of a program. Essentially this is just a special form of oracle access, which has the additional quality that Alice will not learn Bob's input. Nonetheless Bob still needs Alice assistance whenever he wants to evaluate the program on a new input. In order to lift this restriction and enable Bob to evaluate the obfuscated program as many times as he would like on any input of his choice, it is necessary to make some changes. Specifically, instead of using oblivious transfer which lets Bob choose exactly the set of matrices $\{\tilde{A}_{i,\chi_{inp(i)}} : i \in [n], inp(i) > |C'|\}$ that correspond to one specific input x which he chooses, he gets all of the matrices that correspond to his part of the input *without* having to fix any input. The obfuscated program then consists of the set of fixed matrices $\{\tilde{A}_{i,\chi_{inp(i)}} : i \in [n], inp(i) \leq |C'|\}$ belonging to Alice's part of the input as well as the complete set of matrices corresponding to Bob's part of the input $\{\tilde{A}_{i,b} : i \in [n], b \in [0,1], inp(i) > |C'|\}$. This will allow Bob to choose the correct matrices for his part of the input whenever he would like to evaluate the program.

3.2 Problems and Solutions

While the plain version of Killian's protocol is secure, that is Bob is not able to gain any information about the evaluated circuit or Alice's parts of the input, the alterations described above give Bob substantially more information which he can use to mount attacks. In particular there are three categories of attacks which should capture all potential attacks: attacks that ignore the algebraic structure, partial evaluation attacks and mixed input attacks. The following sections describe the three types of attacks as well as the techniques that can be used to mitigate them.

3.2.1 Attacks Not Respecting the Algebraic Structure

If an attacker ignores the given matrix structure or computes non-multilinear algebraic functions over the matrices, he might be able to mount some kind of non-trivial attack on the described scheme. In the original paper [1] the authors mention that computing matrix inverses which have an algebraic degree of -1 can potentially lead to such kind of attack, without further discussing how such an attack would look like. In order to prevent these attacks, it is possible to use Multilinear Jigsaw Puzzles 2.6 in order to encode each matrix, thus making non-multilinear attacks infeasible. This forces any possible attack into one of the other two categories.

Essentially the product of matrices $P = \prod_{i\in[n]} \tilde{A}_{i,\chi_{inp(i)}}$ that will occur when Bob wants to evaluate the program on one input of his choosing, can already be understood as a valid multilinear form for the Multilinear Jigsaw Puzzle. By encoding the elements of each matrix $\tilde{A}_{i,b}$ at level $\{i\} \subseteq [n]$ it is already guaranteed that only elements which are encoded at different levels are multiplied, while only elements which are encoded at the same level will be summed. To demonstrate let us assume we have the product of two 2x2 matrices:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a\cdot e + b\cdot g & a\cdot f + b\cdot h \\ c\cdot e + d\cdot g & c\cdot f + d\cdot h \end{pmatrix}$$

Now we encode each element of the first matrix at level $\{0\}$ and each element of the second matrix at level $\{1\}$. Using $E_s(x)$ as a shorthand for an encoding of element x at level s, we get:

$$\begin{pmatrix} E_{\{0\}}(a) & E_{\{0\}}(b) \\ E_{\{0\}}(c) & E_{\{0\}}(d) \end{pmatrix} \cdot \begin{pmatrix} E_{\{1\}}(e) & E_{\{1\}}(f) \\ E_{\{1\}}(g) & E_{\{1\}}(h) \end{pmatrix} = \begin{pmatrix} E_{\{0\}}(a) \cdot E_{\{1\}}(e) + E_{\{0\}}(b) \cdot E_{\{1\}}(g) & E_{\{0\}}(a) \cdot E_{\{1\}}(f) + E_{\{0\}}(b) \cdot E_{\{1\}}(h) \\ E_{\{0\}}(c) \cdot E_{\{1\}}(e) + E_{\{0\}}(d) \cdot E_{\{1\}}(g) & E_{\{0\}}(c) \cdot E_{\{1\}}(f) + E_{\{0\}}(d) \cdot E_{\{1\}}(h) \end{pmatrix}$$

As one can see, every element that is encoded at level $\{0\}$ is multiplied with an element at level $\{1\}$. Following the rules for the evaluation of Multilinear Jigsaw Puzzles, multiplying two elements of different level sets together will create a new element that is encoded at the union of the two levels. During the addition that follows the multiplication at both sides, both summands will then be encoded at the level $\{0,1\}$. This is again a valid operation, because the requirement for addition is that all summands are encoded at the same level. In the end we will be left with a matrix each of which elements are encoded at level $\{0,1\}$.

When encoding a branching program of length n, one should use a Multilinear Jigsaw Puzzle with multilinearity also equal to n. This ensures that the product that arises from the evaluation of the branching program will be encoded at the highest level $[n]$, which is convenient as this is required by the zero testing procedure. Of course, it is not useful to be able to test if the product is zero, but this problem will be dealt with below. This encoding prevents non-multilinear attacks.

3.2.2 Partial Evaluation Attack

In the process of giving Bob the capability of evaluating the obfuscated program for multiple outputs of his choosing, he inadvertently also gained the power to compare some of the intermediate stages of the evaluation process. To do this Bob chooses $j, k \leq n$ and two different inputs x, x' for his part of the input leading to $\chi, \chi^{'}$ for the joint input and computes $\prod_{i=j}^{k} \tilde{A}_{i,\chi_{ind(i)}}$ and $\prod_{i=j}^{k} \tilde{A}_{i,\chi'_{ind(i)}}$. Notice that the outer random matrices R_{j-1} and R_k^{-1} will be the same for both products, which means that if they are equal for two different inputs, the probability is extremely high that the partial evaluation of the not-randomized branching program indeed yields the same result for the inputs χ and χ'. This can be used to extract information about the program that the obfuscator needs to hide.

Imagine for example that Bob is able to choose j and k in such manner that he knows that the instructions between them correspond exactly to the evaluation of one gate of the original circuit that should be obfuscated. This is certainly possible since the structure of the Universal Circuit used in the construction is not kept secret and Bob could generate the Branching Program for that circuit himself. By cleverly crafting his inputs he could control the inputs to the gate in question and subsequently compare the outputs, which would allow him to determine the type of that gate. To establish if the gate is an AND- or an OR-gate, he could for example choose an input which causes the two inputs 01 and 11 to the gate that he is interested in. If they yield the same result it is an OR-gate, if the result is different it is an AND-gate. Using this technique he would in the end be able to recover the whole circuit, which obviously breaks the definition of security for an indistinguishability obfuscator.

"Bookends" and Higher Dimensional Matrices To counter this kind of attack, it is necessary to prevent Bob from comparing partial computations or to be more precise to prevent him from gaining information by comparing them. To do so special 'bookend' components will be added to the branching program, without which it is impossible to evaluate any computation. The first step is to embed all of the $A_{i,b}$ into bigger matrices like so:

$$D_{i,b} \sim \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & A_{i,b} \end{pmatrix}, \qquad \tilde{D}_{i,b} \sim R_{i-1} \cdot \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & A_{i,b} \end{pmatrix} \cdot R_i^{-1}$$

In these matrices all unspecified entries are 0 and the \$ are random elements from $\mathbb{Z}_p$. The size of the diagonal of random elements shall be $2m$ (m will be specified below). Just as in the original protocol by Killian the R_i are random invertible matrices over $\mathbb{Z}_p$ and we obtain the $\tilde{D}_{i,b}$ by applying them as seen above.

Now when evaluating the program for a particular input χ, one can calculate $\prod_{i\in[n]} \tilde{D}_{i,\chi_{inp(i)}} = R_0 \cdot P \cdot R_n^{-1}$, where P is some high dimensional matrix which has the result of the branching program in its lower right $5x5$ quadrant. In order to obtain the actual result, the two 'bookend' vectors s and t with dimensions $2m+5$ equal to that of the matrices are generated:

$$s \sim (0...0\ \$...\$ - s^* -), \qquad t \sim (\$...\$\ 0...0 - t^* -)$$

The first two blocks shall have a size of m and s^* and t^* are both of size 5. Accordingly $\tilde{s} = s \cdot R_0^{-1}$ and $\tilde{t} = R_n \cdot t$. Additionally the scalar $p^* = \langle s^*, t^* \rangle$ will be saved with the obfuscation.

In order to also be able to encode the 'bookend' vectors, the obfuscator increases the multilinearity of the Multilinear Jigsaw Puzzle to $n+2$ and encodes $\tilde{s}$ and $\tilde{t}$ at the levels $\{1\}$ and $\{n+2\}$ respectively. Accordingly the matrices $\tilde{D}_{i,b}$ shall be encoded at level $\{i+1\}$ instead of $\{i\}$ and the scalar p^* at the highest level $[n+2]$. Now one can add these vectors at the start and the end of the computation:

$$p = \tilde{s} \cdot \left(\prod_{i \in [n]} \tilde{D}_{i, \chi_{inp(i)}} \right) \cdot \tilde{t}^T = \tilde{s} \cdot (R_0 \cdot P \cdot R_n^{-1}) \cdot \tilde{t}^T = s^* \cdot A \cdot t^{*T}$$

Here A is the result of the branching program. Since all these calculations are actually performed in the encoding and all $n + 2$ elements are multiplied together, p will be encoded at the highest level $[n + 2]$. This means one can calculate $p' = p - p^*$ as p and p^* are encoded at the same level. If A is the identity matrix it holds that $p = s^* \cdot A \cdot t^{*T} = \langle s^*, t^* \rangle = p^*$ and therefore $p' = 0$, which is convenient since this is exactly what can be tested with the Jigsaw Verifier. Recall that the original Branching Program is crafted such that the resulting permutation is exactly the identity iff the corresponding circuit evaluates to 1, that is $A_1 = I$. The obfuscated program will return 1 exactly when the Zero-Test on p' returns True, which is correct because the probability is very small that $p' = 0$ if A is any other permutation matrix.

3.2.3 Mixed Input Attacks

In all but the most simple of branching programs, there will be several instructions that inspect the same input bit. Say instruction i and j both look at the same input bit k, that is $inp(i) = inp(j) = k$. During a correct evaluation one should either use $A_{i,0}$ and $A_{j,0}$ if $\chi_k = 0$ or otherwise $A_{i,1}$ and $A_{j,1}$. In a mixed input attack the attacker does not stick to these rules and uses $A_{i,0}$ together with $A_{j,1}$ or vice versa, which can again lead to him learning some information that he should not be able to learn.

Multiplicative Bundling To ensure that any evaluation that mixes instructions as described above only produces useless output, one can add multiplicative factors to all of the $A_{i,b}$. Namely Alice chooses scalars $\{\alpha_{i,b} : i \in [n], b \in \{0, 1\}\}$ randomly and applies them like so:

$$D_{i,b} \sim \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & \alpha_{i,b} \cdot A_{i,b} \end{pmatrix}, \qquad \tilde{D}_{i,b} \sim R_{i-1} \cdot \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & \alpha_{i,b} \cdot A_{i,b} \end{pmatrix} \cdot R_i^{-1}$$

The first consequence of this is, that it is not enough any more to know p^* to be able to decode the final result of the computation. As a remedy Alice creates a 'dummy program' which has basically the same structure as the original program, but which computes the constant 1 function and thus has only the identity permutation for every $A_{i,b}$ of the original program. She generates a second set of matrices R_i', two vectors s', t' of the same shape as s and t where $\langle s', t' \rangle = \langle s, t \rangle$ and consequently computes $\tilde{s}' = s' \cdot R_0'^{-1}$ and $\tilde{t}' = R_n' \cdot t'$. Additionally she chooses a second set of alphas $\{\alpha_{i,b} : i \in [n], b \in \{0,1\}\}$ with the constraint that

$$\prod_{inp(i)=j} \alpha_{i,b} = \prod_{inp(i)=j} \alpha'_{i,b} : \forall j \in [l], b \in \{0,1\}$$

Now she can generate all the instructions of the 'dummy program':

$$D'_{i,b} \sim \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & \alpha'_{i,b} \cdot I \end{pmatrix}, \qquad \tilde{D}'_{i,b} \sim R'_{i-1} \cdot \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & \alpha'_{i,b} \cdot I \end{pmatrix} \cdot R_i'^{-1}$$

With all the elements of the 'dummy program' in place, they can be encoded with the Multilinear Jigsaw Puzzle in the same way as the 'primary' program. To evaluate the result of the whole obfuscated program the idea is to evaluate both original and dummy program and test if they yield the same result. Specifically one computes the following formula:

$$\delta = \hat{s} \cdot \left(\prod_{i \in [n]} \hat{D}_{i, \chi_{inp(i)}} \right) \cdot \hat{t}^T - \hat{s}' \cdot \left(\prod_{i \in [n]} \hat{D}'_{i, \chi_{inp(i)}} \right) \cdot \hat{t}'^T$$

Here $\hat{x}$ stands for the encoding of x relative to the corresponding index set as described before. The Jigsaw Verifier can then be used to test if $\delta = 0$, in which case the result of the evaluation is 1.

This multiplicative bundling does indeed prevent mixed input attacks. The constraint on the $\alpha_{i,b}, \alpha_{i,b}i'$ causes them to evaluate to the same product for an honest evaluation. During an attempt to mix and match the matrices, there is only a probability of $1/p$ that the product of $\alpha_{j,0} \cdot \alpha_{k,1} = \alpha'_{j,0} \cdot \alpha'_{k,1}$.

3.3 Complete candidate

In the preceding sections it was established how to harden the basic version of Killian's protocol against the attacks that open up when giving Bob the complete set of matrices belonging to his part of the input. In this section the whole construction is presented in condensed form:

Definition 8 (Constructing Randomized Branching Programs)**.** Initially choose the ring $\mathbb{Z}_p$ over which the input Branching Program shall be randomized. The value of m is $2n + 5$ where n is the length of the Branching Program.

1. Generate random and independent $\{\alpha_{i,b}, \alpha'_{i,b} : i \in [n], b \in \{0,1\}\}$ with the constraint that
$$\prod_{inp(i)=j} \alpha_{i,b} = \prod_{inp(i)=j} \alpha'_{i,b} : \forall j \in [l], b \in \{0,1\}$$

2. Generate matrices $\{D_{i,b}, D'_{i,b} : i \in [n], b \in \{0,1\}\}$ of size $2m+5$ with their diagonals from 1 to $2m$ populated randomly from $\mathbf{Z}_p$. Set the bottom right 5x5 quadrant of the $D_{i,b}$ to $\alpha_{i,b}\dot{A}_{i,b}$ and of the $D'_{i,b}$ to $\alpha'\dot{I}$:
$$D_{i,b} \sim \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & \alpha_{i,b} \cdot A_{i,b} \end{pmatrix}, \quad D'_{i,b} \sim \begin{pmatrix} \$ & & & \\ & \ddots & & \\ & & \$ & \\ & & & \alpha'_{i,b} \cdot I \end{pmatrix}$$

3. Randomly draw four vectors of length 5 $\{s^*, t^*, t'^*, s'^*\}$ so that $\langle s^*, t^* \rangle = \langle s'^*, t'^* \rangle$ and assemble vectors of length $2m+5$ $\{s, t, s', t'\}$ where \$ are random elements from $\mathbf{Z}_p$:
$$\begin{array}{ll} s \sim (0...0 \ \$...\$ - s^* -), & t \sim (\$...\$ \ 0...0 - t^* -)^T \\ s' \sim (0...0 \ \$...\$ - s'^* -), & t' \sim (\$...\$ \ 0...0 - t'^* -)^T \end{array}$$

4. Sample random full rank invertible matrices $\{R_i, R'_i : i \in [n]\}$ of size $(2m+5) \times (2m+5)$

5. The complete Randomized Branching Program is:

$$\mathcal{RBP}_p(BP) =$$
$$\left\{ \begin{array}{ll} \tilde{s} = sR_0^{-1}, \tilde{t} = R_n t, & \tilde{s}' = s'(R'_0)^{-1}, \tilde{t}' = R'_n t', \\ \{\tilde{D}_{i,b} = R_{i-1} D_{i,b} R_i^{-1} : i \in [n], b \in \{0,1\}\}, & \{\tilde{D}'_{i,b} = R'_{i-1} I (R'_i)^{-1} : i \in [n], b \in \{0,1\}\} \end{array} \right\}$$

As already mentioned in the derivation, this are essentially two programs, the dummy and the original one. During evaluation one evaluates both of them and outputs 1 when they have the same result.

Definition 9 (Candidate Indistinguishability Obfuscator for $\mathbf{NC}^1$ Circuits)**.** Given a circuit C with l inputs and s gates, one proceeds as follows:

1. Generate the Universal Circuit $UC_{s,l}$

2. Generate the corresponding Universal Branching Program $UBP_{s,l}$ using Barrington's Theorem

3. Use the Jigsaw Specifier with some security parameter λ and the length of $UBP_{s,l}$ as multilinearity parameter and get as output the prime p

4. Generate the Randomized Branching Program $\mathcal{RBP}_p(UBP)$

5. Encode the Randomized Branching program like so:

$$\widehat{\mathcal{RBP}}_p(UBP) = \left\{ \begin{array}{ll} \hat{s} = \mathsf{Encode}_{\{1\}}(\tilde{s}), \quad \hat{t} = \mathsf{Encode}_{\{n+2\}}(\tilde{t}), & \hat{s}' = \mathsf{Encode}_{\{1\}}(\tilde{s}'), \quad \hat{t}' = \mathsf{Encode}_{\{n+2\}}(\tilde{t}'), \\ \{\hat{D}_{i,b} = \mathsf{Encode}_{\{i+1\}}(\tilde{D}_{i,b}) : i \in [n], b \in \{0,1\}\}, & \{\hat{D}'_{i,b} = \mathsf{Encode}_{\{i+1\}}(\tilde{D}'_{i,b}) : i \in [n], b \in \{0,1\}\} \end{array} \right\}$$

6. For every input $\chi \in \{0,1\}^l$ the corresponding multilinear form will be:

$$\mathcal{F}_\chi(\mathcal{RND}_p(UBP)) = \hat{s}(\prod_i \hat{D}_{i,\chi_{inp(i)}})\hat{t} - \hat{s}'(\prod_i \hat{D}'_{i,\chi_{inp(i)}})\hat{t}' \mod p$$

7. Finally generate a binary description C' of C and fix all the matrices that belong to that part of the input:

$$\mathsf{Fix}(\widehat{\mathcal{RBP}}, C')_p(UBP) = \left\{ \begin{array}{ll} \hat{s}, \hat{t}, & \hat{s}', \hat{t}', \\ \{\hat{D}_{i,C'_{inp(i)}} : i \in [n], inp(i) \leq |C'|\}, & \{\hat{D}'_{i,C'_{inp(i)}} : i \in [n], inp(i) \leq |C'|\}, \\ \{\hat{D}_{i,b} : i \in [n], b \in \{0,1\}, inp(i) > |C'|\}, & \{\hat{D}'_{i,b} : i \in [n], b \in \{0,1\}, inp(i) > |C'|\} \end{array} \right\}$$

The underlying idea for the hardness assumption of this construction is that it is not feasible to distinguish two different partial assignments of matrices of the encoded Universal Randomized Branching Program, if these assignments cause it to compute the same function. This is exactly the requirement for an indistinguishability obfuscator. For a formal proof of the hardness assumption we refer to [1] Appendix C.

3.4 Limitations

The construction above is only a valid indistinguishability obfuscator for circuits in $\mathbf{NC}^1$, because of the polynomial slowdown requirement for obfuscators. The obfuscation would still work for deeper circuits, but the overhead would not be polynomial anymore. While the described candidate can only handle circuits with one outputs, it would be no problem to repeat the process for every additional output bit.

Garg et al. [1] also provide an amplification method to build an indistinguishability obfuscator for polynomial sized circuits. Besides the obfuscator for $\mathbf{NC}^1$ circuits, it also makes use of Perfectly Binding Commitments [25], Perfectly Sound Non-Interactive Witness Indistinguishability Proofs and Fully Homomorphic Encryption [26]. All of them are not trivial cryptographic schemes and especially Homomorphic Encryption is by no means ready for practical use. Especially considering that most of these constructs are needed in circuit form, because the $\mathbf{NC}^1$ candidate is applied on top of them, we decided that the construction of the indistinguishability obfuscator for polynomial sized circuits is out of the scope for this thesis. For more details on the construction we refer to the original paper.

4 Implementation

This chapter discusses how we implemented the mathematical constructs which are needed for the construction of the indistinguishability obfuscator candidate. All of the code is written in Python, using the libraries of the mathematical software Sage [2] for the more advanced algebraic concepts, most notably the quotient rings over $\mathbb{Z}_p$ used by the Multilinear Jigsaw Puzzles. Sage and the libraries that it uses are partially implemented in C, but it offers easy integration into Python programs. Apart from that the implementation only uses some modules from the python standard library as needed and two small modules from the Python Package Index [27], one implementing topological sorting and another one for timing the duration of commands. We are using Sage version 6.4 and the embedded Python version 2.7.8.

4.1 Circuits

In the implementation, circuits are essentially a recursive data structure, similar to a binary tree only that not all nodes have two inputs. Instead of nodes it consists of gates, which can be of type `AndGate` ($\wedge$),`NotGate` ($\neg$) or `Input`. `Input` gates are similar to the leaves of a binary tree in that they do not have any further references to other gates. Each `Input` has a variable `pos`, which stores the position of its assigned input bit in the input string. `AndGates` are the equivalent of normal nodes in a binary tree and each references two other gates of any type. These references can be accessed via the variables `inp1` and `inp2`. `NotGates` are the new addition that make these circuits slightly different from binary trees, because they only reference exactly one other gate, which can be accessed via the variable `inp`. The references that each gate holds are exactly the inputs over which they compute their function.

The circuit object itself only maintains a reference to its output gate and a list with all the gates of type `Input` which are located in itself. To evaluate a circuit, the list with `Input` gates is used to assign a value to each of them, then the evaluation begins recursively with the output gate of the circuit. When a gate is evaluated, its behavior depends on its type. An `Input` gate simply returns the value that it was assigned, while a `NotGate` first evaluates its input and then returns the opposite of that value. `AndGates` evaluate both of their inputs, before computing the logical AND of both values and returning the result.

4.2 Universal Circuits

There exist several different methods of constructing Universal Circuits, with an asymptotically optimal given already by Valiant [28]. His construction is based on graphs and graph embedding and with all optimizations achieves a size in $O((s+l)\log s)$ for $UC_{s,l}$, but this is far from trivial to implement.

Additionally there exists another construction due to Schneider [29] which achieves a size in $O(s \log s^2 + (s+l)\log l)$ but offering much smaller constant factors, making it the better choice for small circuits. There even exists an implementation of this construction [30], but it was not possible to integrate it, because the notion of gates used in that implementation is a bit more powerful than what we have available here. Schneider also presents an simple way of constructing Universal Circuits based on building blocks, which is the method that we choose to use, since the Universal Circuit is not the focus of this thesis. This particular method has a size in $O(s^2 + sl)$.

4.2.1 Construction Based on Building Blocks

This explanation closely follows [29], but contains several adaptations in particular the addition of control inputs and the instantiations of Simulation and One-Output Switching Blocks. The basic idea of this construction is the natural method behind any Universal Circuit construction and also employed in the other two methods. It uses some kind of simulation block for each gate G_i in the original circuit (with $i \in [s]$) and than proceeds to hide the original wiring, by making sure that every possible wiring could actually be enabled in the Universal Circuit.

Notation A block B^u_v is a circuit with u inputs $x_1, ..., x_u$ and v outputs $y_1, ..., y_v$. B^u_v computes a function $f_B : \{0,1\}^u \to \{0,1\}^v$ which maps some inputs to some outputs. A programmable block $B^{c,u}_v$ has u inputs and v outputs, but also comes with c control inputs which determine what kind of function the block computes. While the u inputs can be the outputs of another block, the control inputs will be native inputs to the resulting Universal Circuit constructed out of the programmable blocks that are defined below. All blocks in this section will only be created out of AND and NOT gates, in order to facilitate the application of Barrington's Theorem 1 during the transformation of Universal Circuit to Branching Program.

Simulation Block A simulation block G^{Sim} can simulate one gate of the original circuit and has two inputs x_1, x_2, one control input c and one output y. Depending on the value of the control input it outputs simply $\neg x_1$ for $c = 0$ or $x_1 \wedge x_2$ for $c = 1$. This is implemented as $\neg(x_1 \wedge \neg x_2) \wedge \neg(x_1 \wedge \neg c) \wedge \neg(\neg x_1 \wedge c)$ which, if we check the truth table, indeed has the desired behavior.

One-Output Switching Block A one-output switching block Y is a programmable block with two inputs x_1, x_2, one control input c and one output y. It outputs either x_1 or x_2 depending on the value of the control input. It is implemented as $\neg(\neg c \wedge \neg x_1) \wedge \neg(c \wedge \neg x_2)$. Again the truth table verifies the correctness of this.

x_1	x_2	c	$\neg(\neg c \wedge \neg x_1)$	$\neg(c \wedge \neg x_2)$	y
0	0	0	0	1	0
1	0	0	1	1	1
0	1	0	0	1	0
1	1	0	1	1	1
0	0	1	1	0	0
1	0	1	1	0	0
0	1	1	1	1	1
1	1	1	1	1	1

Table 4.1: Truth table of one-output switching block

Selection Blocks A selection block S_v^u selects one of its u inputs for each of the v outputs without any restrictions, making it possible to select the same value several times or to change the order. It is quite straightforward to construct S_1^u blocks by using $(u-1)$ Y blocks either arranging them in a tree or chaining them together. In fact note that a single Y block is already a S_1^2 block. Considering that each Y block has one control input, it follows immediately that an S_1^u block has $(u-1)$ control inputs. When the Y blocks are arranged as in Figure 4.1, one can also easily see that if one wants to select input i, it is sufficient to set control input $i-1$ to 1 and leave all other control inputs at 0 to do so. By doing so one causes the corresponding Y block to select its right input, while leaving the others at 0 causes them to pass the selected values up to the output. To select the first input simply leave all control inputs at 0. A simple method of constructing a general S_v^u block uses one S_1^u block for each of the v inputs. This is not very efficient and there are better constructions presented in [29], but we will use this construction.

Universal Block A universal block U_k is a special type of Universal Circuit for k gates, which has some specific requirements on the inputs and outputs of circuits that it can simulate (so it is not actually universal). Particularly U_k has $2k$ inputs and k outputs,

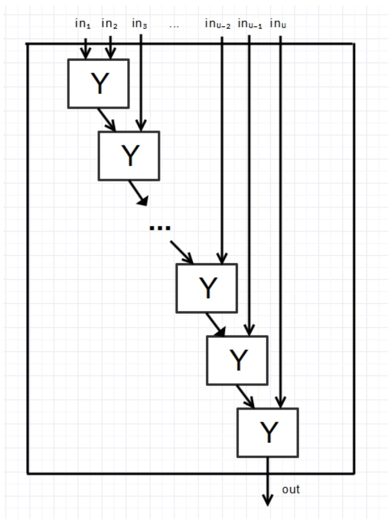

Figure 4.1: S_1^u selection block construction using chaining

which is the maximum that a circuit with k gates can have (it would consist exclusively of k AND gates). The requirements on the inputs are that the inputs x_{2i-1}, x_{2i} are exactly and only going to the gate G_i. This does not mean that G_i has to use these inputs, it can also pick any other G_j with $j < i$ as input, just not another natural input to the circuit. The requirement on the outputs is that each output i can only come from gate G_i, but because we only have to consider circuits with exactly 1 output, we will drop all but the last output in our construction.

Figure 4.2 shows the structure of a U_k block. Essentially there exists a simulation block G_i^{Sim} for each gate G_i of the original circuit and for each of the G_i^{Sim} there are two selection blocks S_1^i, one for each of the simulation blocks inputs. These selection blocks can either choose the input x_{2i-1} (x_{2i} if it is the second one) or the output of any of the $(i-1)$ simulation blocks that came before this one. Which gate of the original circuit shall be simulated by which simulation block is determined by doing a topological sorting, which ensures that the inputs to every G_i are either native inputs to the circuit or a G_j with $j < i$. More detail on that can be found in Section 4.2.2.

Universal Circuit Given methods to build a universal block and general S_v^u selection blocks, it is simple to construct true Universal Circuits $U_{s,l}$ by chaining them as showed in Figure 4.3. The upper S_{2s}^l makes it possible to heed the input restriction of the U_s block, since one can direct each input to wherever it is needed. Usually one would also use an output selection block, but the candidate construction only works for circuits with exactly one output, hence it is sufficient to use the output of the last simulation

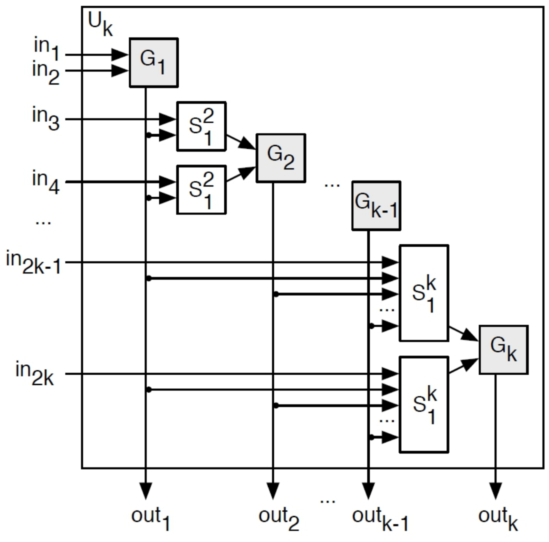

Figure 4.2: Universal Block construction (source: [29])

gate G_s^{Sim} as the output of the whole Universal Circuit.

4.2.2 Generating the Binary Description of a Circuit

In order to generate the control input C' of a circuit C, it is first necessary to make a topologically sorted list of all the gates in C. In such a sorting, the first gates will be the ones without any dependencies, i.e. the ones which directly use the native inputs to the circuit. Following after them will be the gates that take these first gates as inputs and so on, until the output gate of the circuit, which will come last, since every other gate needs to be evaluated before it. Since the circuits are acyclic, such a topological sorting can be obtained easily. In order to prevent ambivalence, we use the internal unique ID of each gate as a tiebreaker, meaning that earlier created gates will come first. The position of each gate in the topological order defines by which simulation block that gate shall be simulated.

For each simulation block we need to determine which kind of gate it needs to simulate and what values the two selection blocks shall feed to its inputs. To determine this, we enumerate all gates in the order defined by the topological sorting and do the following:

1. Check the gate type to get the control input to the simulation block (0 for `noGates`, 1 for `andGates`)

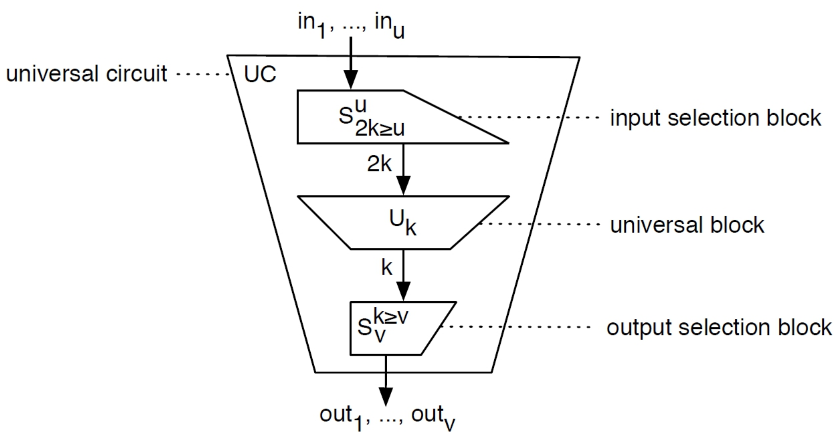

Figure 4.3: Universal Circuit construction (source: [29])

2. Look at the first input to that gate:
   - if it is an `Input`: let the selection block select the first value (control inputs all 0) and wire the input with the 2^l_{2s} selection block to input $2i-1$ of the U_s block
   - if it is another gate: check where in the topological sorting that gate is (it must have come before) and select that simulation block (if gate is at position x in topological order set control input x of the selection block to 1)
3. If the gate is an `andGate`, repeat step 2 for the second input, otherwise leave the second selection block at default (control inputs all 0)

This gives us the correct values to all of the control inputs. To make sure that each control input receives the correct value as intended, the order of the control inputs of each Universal Circuit is defined like so: First the $2s(l-1)$ control inputs of the first S^l_{2s} block that distributes the native inputs correctly to the U_s block. Afterwards for each G^{Sim}_i there follow the control inputs to the upper S^i_1 selection block, then the control inputs to the lower one and finally the control input to the G^{Sim}_i itself. Using this method we can use the control inputs to change the wiring of the Universal Circuit, so that it can compute the same function as the original circuit. The length of a binary description of a circuit with s gates and l inputs is $length_{Description}(s,l) = 2s(l-1) + 2\sum_{i=1}^{s}(i-1) + s = s^2 + 2ls - 2s$.

4.3 Circuit to Branching Program

Recall Definition 1 about Branching Programs, who are essentially a sequence of instructions with each instruction having three parts, namely the value of $inp(i)$ telling us which input bit to look at and two permutations. During the implementation it turned out to be more efficient to maintain three separate lists for the different instruction parts, instead of maintaining one list with all three of them. Thus we have one `indexList` which stores the values of $inp(i)$ for $i \in [n]$ and two lists for permutations, the first one called `ins0` stores all $A_{i,0}$ and the second one `ins1` all $A_{i,1}$. In Section 2.4 we already established that these permutations can also be understood as 5×5 permutation matrices $A_{i,b}$, which is in fact how they are internally represented, but for ease of notation we will continue to refer to them as permutations for most of this section. The permutation matrices are implemented using NumPy [31] arrays, which are slightly more efficient than the matrices provided by Sage. For the evaluation each Branching Program additionally stores two permutations according to the A_0 and A_1 of the mathematical definition. If the result of multiplying all permutations corresponding to one specific input is equal to A_0 the output is 0, if it is equal to A_1 it is 1.

At the heart of the circuit to branching program transformation lies a recursive function which implements the inductive proof of Barrington's theorem. This function operates on the gates of the circuit that shall be transformed and calculates the contents of the three lists needed for a Branching Program. In order to make this process as straight forward as possible, it always returns a 'normalized' Branching Program, that is a Branching Program with $A_0 = e$ and $A_1 = (1\ 2\ 3\ 4\ 5)$, which is possible due to Lemma 2. This accomplishes that the permutations that need to be applied for each gate type are always the same and no expensive computations have to be made. While the general process of this procedure was already explained in Section 2.3.1, the concrete values used for Lemma 2 where omitted. Recall that for any two 5-cycles σ, τ it is possible to find a φ so that $\tau = \varphi\sigma\varphi^{-1}$. These are the values for φ that we use during the construction:

- Changing the inverse of the normal permutation $(5\ 4\ 3\ 2\ 1)$ to the normal permutation $(1\ 2\ 3\ 4\ 5)$ as needed for a `notGate`: $(2\ 5)(3\ 4)$ This permutation is also used for the inverse of this switch which is needed to compute P_0' for an `andGate`.
- Changing the normal permutation to the second permutation $(1\ 3\ 5\ 4\ 2)$ as required to compute P_1 for an `andGate`: $(2\ 5\ 3)$
- Changing the second permutation to its inverse, required for P_1': $(2\ 3)(4\ 5)$ This permutation is also used to normalize the result of an `andGate` $(1\ 3\ 2\ 5\ 4)$.

After the recursive function has finished, we multiply the inverse of the normal permutation to both rightmost instruction. This causes the Branching Program to have $A_1 = e$ as required by the rest of the construction.

During testing we discovered that the straightforward adaption of Barrington was quite slow and extremely memory consuming, mostly because of the recursive creation of large lists containing NumPy arrays and the many multiplications that are needed for changing and normalizing the Branching Programs in each step. To speed this up, we decided not to represent the permutations as matrices during the generation but to assign each permutation a unique integer id which could be mapped to matrices after the generation. Since there are only 120 different 5×5 permutation matrices (permutations over S_5), having such a mapping is no problem.

On these integer `ID`s, a pass of Lemma 2 can be understood as a mapping from one `ID` to another, which depends on the value of φ. We first combined the permutations, where more than one pass of Lemma 2 was applied to the same sequence of instructions, which happened every time a Branching Program was first changed and then normalized. This brings down the number of required mappings to six, namely two for `notGates` (change the last instruction and normalize the rest of the program) and four for `andGates` (one each for generating P_0, P_1, P_0' and P_1'). To be able to perform these mappings quickly we calculated them in advance and have them saved in a hard coded and quickly accessible lookup structure. Using this technique we were able to cut down the time needed to generate Branching Programs by a factor of 13 and memory usage during generation by even more.

Reuse Problem One big problem with Branching Programs is that they do not handle reused gates very well. In a circuit it is no problem to take the output of one gate g and wire it to several other gates, to evaluate the circuit it is still sufficient to evaluate g just once, as all of the other gates can reuse the output. However, when building a Branching Program, the part that calculates g needs to be duplicated whenever the output of g is used by another gate, because there is no way to share the result. The impact of this is especially high for the Universal Circuits that we construct, because they reuse the output of previous gates a lot. For example in the implementation of simulation blocks the input x_1 is reused three times, which means that the whole circuit that feeds into x_1 will be included three times in the corresponding Branching Program. The same goes for each S_1^i block, with each of them using the outputs of all of the preceding G_j^{Sim} blocks with $j < i$. Since this effect stacks up and parts that reuse other parts get reused as well, the size of the Branching Programs for Universal Circuits grows extremely fast.

4.4 Randomized Branching Programs

Similarly to Branching Programs, Randomized Branching Programs 8 also have an `indexList` and two lists, one for the matrices $\tilde{D}_{i,0}$ and one for $\tilde{D}_{i,1}$ respectively. Since they essentially contain two programs, the original one and a dummy one, there also need to be two additional lists that contain the matrices $\tilde{D}'_{i,0}$ and $\tilde{D}'_{i,1}$. Recall that these matrices do not contain regular integers but elements of $\mathbb{Z}_p$. This is where we use Sage for the first time, because it offers a solid implementation not only of $\mathbb{Z}_p$ but also of matrices over $\mathbb{Z}_p$. To make use of this functionality the NumPy arrays of the branching programs are replaced by the matrices offered by Sage and all matrix elements are coerced into $\mathbb{Z}_p$. Sage also offers methods for generating random elements and even matrices from a ring, which are used for filling the diagonal of the $D_{i,b}$ and for choosing the R_i.

4.5 Multilinear Jigsaw Puzzles

This section only gives an overview over the general process of implementing Multilinear Jigsaw Puzzles and the necessary algebraic structures. A more in depth treatment can be found in [1], Appendix A.

When generating a Multilinear Jigsaw Puzzle instance, the input consists of the multilinearity parameter k and the security parameter λ and the procedure is as follows:

1. Initially choose a large random prime q and a dimension parameter t which should be a large enough power of 2. Large is meant as q being of size approximately $2^{O(t^\epsilon)}$ and t of size $k^{1/\delta}$ where δ, ϵ are some constants $0 < \delta < \epsilon < 1$. The following computations will be done in the rings $R = \mathbb{Z}[x]/(x^t + 1)$ and $R_q = \mathbb{Z}_q[x]/(x^t + 1)$.

2. Choose a small polynomial $g \in R$ so that $|R/(g)|$ is a prime $p > 2^\lambda$. Here small means that all coefficients should be smaller than $2^{O(t^\delta)}$. g also has to fulfill some additional technical conditions, but all of these conditions can be heeded with high enough probability when choosing small polynomials randomly.

3. Sample k random polynomials $z_1, ..., z_k \in R_q$.

The plaintext space for this Multilinear Jigsaw Puzzle will be $\mathbb{Z}_p$. To encode an element a at level $S \subseteq [k]$ perform the following computation:

$$\mathsf{Encode}_S(a) = \frac{\hat{a} + e \cdot g}{\prod_{i \in S} z_i}$$

Here $\hat{a}$ is obtained by computing $a \mod g$ and should be a small polynomial in R. The polynomial e is an error term which also needs to be small like g. Because it is chosen at random, it causes the whole encoding to be randomized. It is critical for the security of the construction.

Finally to allow the Jigsaw Verifier to test if a puzzle was solved (meaning that a multilinear form evaluated to an encoding of 0 at the highest level [k]), the last step of the setup is to generate a zero test element:

$$\mathsf{p}_{zt} = \frac{h \cdot \prod_{i=1}^{k} z_i}{g}$$

Here h should be a random medium sized polynomial with coefficients of size proximately $q^{2/3}$. To perform a zero test for a given element u the Jigsaw Verifier multiplies u with p_{zt}. If u is indeed an encoding of 0 at the highest level, the following holds:

$$u \cdot \mathsf{p}_{zt} = \frac{e \cdot g}{\prod_{i=1}^{k} z_i} \cdot \frac{h \cdot \prod_{i=1}^{k} z_i}{g} = e \cdot h$$

Because of the way that h and g where generated, $e \cdot h$ has an euclidean norm smaller than $q^{7/8}$, which does not hold when u encodes anything else. Thus if this is the case for $u \cdot \mathsf{p}_{zt}$, the Jigsaw Verifier outputs 1.

These encodings are already sufficient to enable additions and multiplications like they are required for a Multilinear Jigsaw Puzzle, just observe the following:

Addition Given $a, b \in \mathbb{Z}_p$ and their encodings $\alpha = \mathsf{Encode}_S(a), \beta = \mathsf{Encode}_S(b)$ at the same level $S \subseteq [k]$ their sum is the following:

$$\alpha + \beta = \frac{(\hat{a} + \hat{b} + e' \cdot g)}{\prod_{i \in S} z_i}$$

Of course the error term grows, but with a right choice of parameters it is possible to enable a polynomial number of additions without it getting out of hand.

Multiplication Given $a, b \in \mathbb{Z}_p$ and their encodings $\alpha = \mathsf{Encode}_A(a), \beta = \mathsf{Encode}_B(b)$ at two disjoint level sets $A, B \subseteq [k]$ their product is the following:

$$\alpha \cdot \beta = \frac{(\hat{a} \cdot \hat{b} + e' \cdot g)}{\prod_{i \in A \cup B} z_i}$$

Again, with the right choice of parameters it is possible to prevent the error term e' from growing too large. Observe how this also already prevents addition at different levels or multiplications with overlapping level sets to produce anything useful. Of course the computation would be possible, but the output would just look random and it would not be possible to perform a zero test in the end.

4.5.1 In Practice

Even though Sage does support polynomial quotient rings, the implementation of the procedure described above was far from straightforward. The biggest uncertainty was the computation of $\hat{a}$. We were not able to find an algorithm which takes as an input a integer $a \in \mathbb{Z}_p$ and a polynomial $g \in \mathbb{Z}[x]/(x^t + 1)$ and outputs a *polynomial* which is congruent to $a \mod g$. In our implementation we simply set $\hat{a} = a$ which is trivially correct, since technically it is a polynomial, but it seems that this was not the intention of the authors.

Another issue was the choice of parameters. There are several parameters which can be set about which the authors only make relatively vague, sometimes conflicting statements regarding their magnitude and relation to each other. After a lot of adjustments, we managed to make some choices which enable a partially working version, which allows the encoding of elements at different levels, addition and multiplication and a zero test at the highest level. Nonetheless error terms do not work properly, because even after a very small number of additions and multiplications of encoded elements, they grow too large and it is not possible anymore to perform a successful zero test. This means that the complete construction is probably not secure since the error terms are crucial for the security of the Multilinear Jigsaw Puzzle. We were not able to figure out if this is related to an inappropriate choice of parameters or because of the way we are computing $\hat{a}$.

4.6 Fixing Programs

Following the description of the complete candidate in Section 3.3, Alice chooses the correct matrices for her part of the input and then sends all of them to Bob. There is a slight improvement that we would like to propose, which will cut down the size of obfuscated programs and might actually make it harder for Bob to launch any attacks, since he has less informations. The idea is that instead of sending all the fixed matrices to Bob, Alice already multiplies all of the fixed matrices to both matrices in the nearest instruction to the left which is not fixed and subsequently drops the fixed matrices. Observe that this will not change the result of Bobs computation in the slightest, since he was going to perform these multiplications anyways, it will however cut down the size of the obfuscated program by a big margin (see experiments in Section 5.3).

To formalize, in the original setting when Alice has a Universal Branching Program and the binary description C' of a circuit C, she sends to Bob the set of fixed matrices F as well as the set of matrices belonging to his input I:

$$F = \{A_{i,C'_{inp(i)}} : i \in [n], inp(i) \leq |C'|\}, \quad I = \{A_{i,b} : i \in [n], b \in [0,1], inp(i) > |C'|\}.$$

The addition now is that for every $j, k \in [n]$ so that there exists $A_{j,b}, A_{k,b} \in I$ but no $A_{l,b} \in I$ where $j < l < k$, she computes $\bar{A}_{j,b} = A_{j,b} \cdot \prod_{i=j+1}^{k-1} A_{i,C'_{inp(i)}}$ for $b \in [0,1]$. Informally this means that whenever there are any fixed matrices between two instructions, these fixed matrices will be multiplied to both matrices in the first instruction. She then gives to Bob $\bar{I} = \{\bar{A}_{i,b} : i \in [n], b \in [0,1], inp(i) > |C'|\}$ and nothing else. This same technique is also possible for Randomized Branching Programs and even for the encoded versions.

5 Experiments

The experiments are essentially divided into two parts: generation and evaluation. The former tries to establish the difficulty of creating the obfuscated version of a given circuit and investigates which step in the generation process is responsible for how much of the overhead. The latter investigates how the cost of evaluating a given circuit increases as it undergoes the obfuscation process. In total this aims not only to assess the feasibility of using this construction in real-world programs, but also to determine which parts contribute most to the cost of the construction and thus need to be improved the most.

Each experiment was repeated X times and every data point was then averaged over the number of repetitions in order to eliminate possible outliers. The execution time was determined using direct access to the process time via `time.clock()`, which according to the documentation [32] is the correct way of benchmarking Python. To determine the amount of RAM consumed by each step in the construction we used Guppy/Heapy [33], a Python heap profiler, where it was possible and `top` otherwise. All experiments where run on a virtual machine with a CPU with 2.30 GHz and 2 GB of RAM.

5.1 Generation

In order to understand how difficult it is to generate the obfuscation of a given circuit, we would like to benchmark each intermediate step of the complete candidate for several input circuit families, each of which contains circuits with the same number of gates and number of inputs. That is, firstly generate all Universal Circuits up to a certain point, transform all of them into their respective Universal Branching Program, then transform these Branching Programs into Randomized Branching Programs by applying the garbling techniques described in 3.3 and finally encode this using a Multilinear Jigsaw Puzzle. In each step we would load the result of the step before and then perform the necessary operations to reach the next step, meanwhile monitoring the time elapsed as well as the increase in memory usage. It turned out that a complete benchmark is often not feasible, because the growth is just too fast and the generated objects really quickly do not fit into memory anymore. For these cases we opted to at least determine an estimation of the implied overhead.

5.1.1 Universal Circuit Generation

The first step simply measures how long it takes to generate a Universal Circuit from the basic building blocks as described in section 4.2 and how much memory the resulting objects use. Additionally we computed some of the interesting data of these circuits, like how many AND and NOT gates they are comprised of.

There are two ways of counting the number of gates that a circuit has. The most intuitive one is to count the number of unique gates in the circuit. However due to the reuse problem explained in section 4.3 it is more useful for us to count the number of gates that the circuit had if it was not possible to use the output of a gate more than once. This essentially entails duplicating every subcircuit whose output is used by two different gates.

The range of Universal Circuits $U_{s,n}$ that we generated corresponds to all useful pairs of size s and number of inputs n up to a maximum number of four inputs. To determine what pairs can be regarded as useful, consider the following. Given a fixed number of inputs n, the smallest circuit that uses every input, has a size of $n-1$ and consists solely of AND gates. The biggest useful circuit on the other hand without reusing any gates or inputs has size $3n-2$ and consists of the same AND gates as the smallest possible one, but also has NOT gates interspersed at every possible position. To understand the formula notice that for $n=1$ the biggest circuit has one NOT gate, and then for each additional input it is possible to add exactly three gates. Thus every size between $n-1$ and $3n-2$ is useful and we generated exactly them for input sizes $\{1,2,3,4\}$.

5.1.2 Applying Barrington's Theorem

In this step we took the generated Universal Circuits from the fist step and transformed them into Branching Programs. While this step does not have any further parameters which could be tweaked, it turned out to be sensible to only transform some of the circuits, because the Branching Programs for the others are already too big in itself, before even applying the next steps.

Thus, instead of generating them and then determining their properties, we used the following recursive formula to calculate the expected length l of the Branching Program which implements the same functionality as a circuit defined by its output gate g (this should be exact and not an estimation):

$$\mathrm{len}_{BP}(g) = \begin{cases} 1 & \text{if type}(g) = \text{Input} \\ \mathrm{len}_{BP}(g.\mathrm{inp}) & \text{if type}(g) = \neg \\ 2\mathrm{len}_{BP}(g.\mathrm{inp1}) + 2\mathrm{len}_{BP}(g.\mathrm{inp2}) & \text{if type}(g) = \wedge \end{cases}$$

This enables us to also estimate some of the other properties, most notably the size in memory which grows linear with the length of the Branching Program, since it essentially only depends on the length of the instruction lists. By dividing the memory usage by the length of the Branching Programs we got an approximate factor of 24.4 Byte per length, which we used to estimate the size of the Branching Programs that we couldn't generate. Using the same method we got a factor of proximately $2 \cdot 10^{-6}$ seconds per length for the generation time, albeit with a bit higher variance, so this might be a bit less precise. All values that are highlighted are calculations or estimations.

s	l	Universal Circuits			Branching Programs		
		AND gates	NOT gates	memory	length	memory	time
1	2	17	26	18672	461	16 KB	0.002 s
2	2	97	150	41560	53549	1380 KB	0.036 s
2	3	157	250	62360	248429	6.3 MB	0.167 s
3	2	497	770	74432	6211757	151 MB	9.226 s
3	3	797	1270	105192	28817837	705 MB	43.604 s
3	4	1097	1770	133264	119242157	2.9 GB	238 s
4	2	2497	3870	120016	$7.20 \cdot 10^{8}$	17.58 GB	1441 s
4	3	3997	6370	157256	$3.34 \cdot 10^{9}$	81.56 GB	1.85 h
4	4	5497	8870	194496	$1.38 \cdot 10^{10}$	337.50 GB	7.68 h
5	3	19997	31870	218424	$3.88 \cdot 10^{11}$	9.46 TB	215.42 h
5	4	27497	44370	264832	$1.60 \cdot 10^{12}$	39.15 TB	37 d
6	3	99997	159370	288824	$4.50 \cdot 10^{13}$	1.10 PB	2.85 y
6	4	137497	221870	344464	$1.86 \cdot 10^{14}$	4.54 PB	11.8 y
7	3	499997	796870	368392	$5.22 \cdot 10^{15}$	127.32 PB	330 y
7	4	687497	1109370	433200	$2.16 \cdot 10^{16}$	526.80 PB	1368 y
8	4	3437497	5546870	531168	$2.50 \cdot 10^{18}$	61.11 EB	$1.58 \cdot 10^{5}$ y
9	4	17187497	27734370	638264	$2.91 \cdot 10^{20}$	7.09 ZB	$1.84 \cdot 10^{7}$ y
10	4	85937497	138671870	754680	$3.37 \cdot 10^{22}$	822.29 ZB	$2.16 \cdot 10^{9}$ y

Table 5.1: Results of Universal Circuit and Branching Program generation

5.1.3 Randomizing the Branching Programs

Even for the already reduced subset of Branching Programs that we could actually generate in the preceding step, it was not possible to generate the randomized counterparts in all but two cases. This despite the fact that we are using the smallest size for the randomized matrices by setting $m = 1$, which results in 7×7 matrices, instead of $m = 2n + 5$ as originally described. We decided to take this liberty because the original paper introduced the value $2n + 5$ as an "additional safeguard for unanticipated attacks" and notes that there is not any known reason why setting $m = 1$ should be insecure. Moreover the construction is very slow and memory consuming, that much bigger matrices would make it completely unusable. Also, we randomized all programs with a value of $p = 1049$, because bigger values lead to much higher memory usage. To understand the impact that bigger matrices and bigger values of p would have on the generation time, we rand a small experiment where we only converted the smallest Universal Branching Program for 2 inputs and 1 gate. The results for this can be found in Figure 5.1.

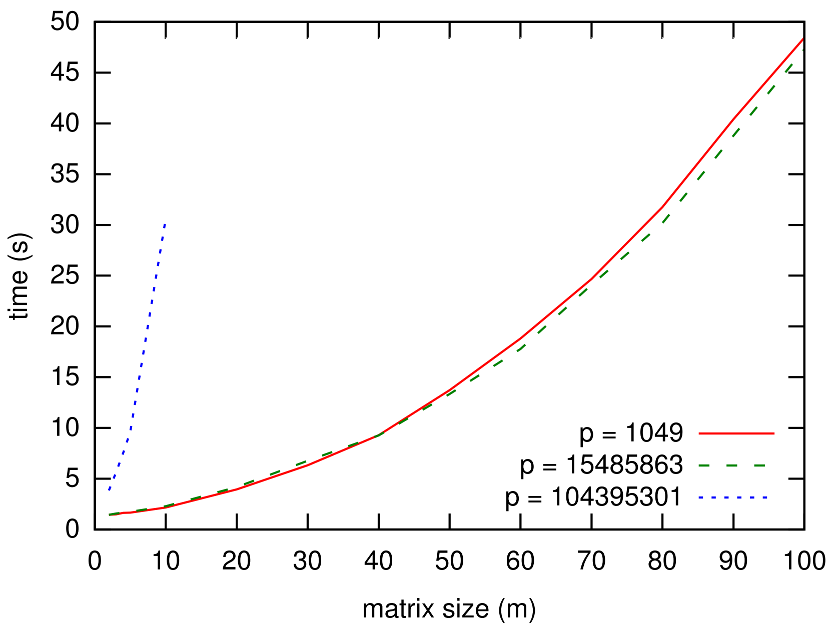

Figure 5.1: Randomized Branching Program Generation with different m

For those Branching Programs where the explicit conversion did not work, we decided to at least try an estimation of the impact of this step in the construction. The principle is again the same, we calculated the factor of Byte per length and seconds per length for the Randomized Branching Programs that we could generate and took these factors to estimate the values for the bigger Branching Programs. Since our sample here is quite small, we also generated some more smaller Branching Programs and randomized them in order to have some more values. All in all we found a cost of about 2500 Byte per length for the memory consumption and $3.5 \cdot 10^{-3}$ seconds per length for the generation time. Notice that this means that Randomized Branching Programs need about 100 times more memory than Branching Programs. The resulting estimations for different Randomized Branching Programs can be found in table 5.2.

A big problem with the data collection for all of the constructs using Sage was that Heapy did not produce reliable results anymore. It turned out that this is due to the fact that it cannot reliably keep track of objects that are generated in some of the libraries that use C/Cython. This meant that we could not collect results in an automated fashion and had to do it by hand using `top` and `ps`.

5.1.4 Encoding with Multilinear Jigsaw Puzzles

Even though our implementation of Multilinear Jigsaw Puzzle is not working completely, as explained in Section 4.5.1, we can use it nonetheless to understand how long it would take to encode the Universal Randomized Branching Programs. Unless correctly computing $\hat{a}$ is very expensive, the results should be very similar for a completely working version. Since we could only create two Randomized Branching Programs, we decided to reach an estimation for the encoding time via a different process than before.

Essentially, instead of measuring the time it takes to encode a whole Randomized Branching Program, we only encoded single elements (that is just one number in the ring $\mathbb{Z}_p$) and noted how long this took. For each set of parameters we repeated this 1000 times for 40 different elements that were randomly sampled from $\mathbb{Z}_p$ and then averaged the results, to get a grasp on how long it takes on average to encode one element. Since we know that a given Randomized Branching Program of length n has $4 \cdot n \cdot (2m+5)^2$ elements (4 sets of $(2m+5) \times (2m+5)$ matrices), we can multiply this with the time it takes to encode one element in order to get an estimation of the time it would take to encode the whole program. As already mentioned above, Heapy does not produce reliable results for Sage object, because it cannot track some of the externally generated objects. It is however possible to treat its results as a lower bound for the increase in memory and that turned out to be a factor of about 3.5 across all of the experiments.

The parameters for this experiment are the number of levels k that the Multilinear Jigsaw Puzzle should support and the dimension t which determines the degree of the polynomials that represent encoded elements. The number of levels should actually be the length of the Branching Program, so here it is useful to look a bit at the development of the encoding time dependent k. The dimension on the other hand is defined in the paper as $t = k^{1/\delta}$, which is simply not possible for practical purposes. Thus we decided to look at a number of fixed dimensions instead. The encoding performed is always an encoding at the highest level, since that is the most difficult one requiring

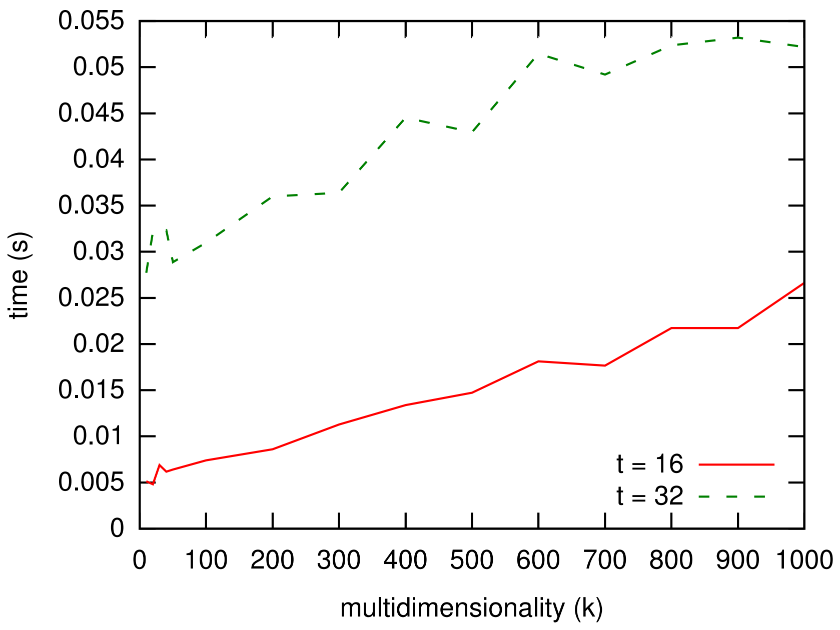

Figure 5.2: Encoding time development with increasing multilinearity

the maximum number of multiplications, thus giving an upper bound on the difficulty.

The results in Figure 5.2 show that the encoding times grow approximately linear with the multidimensionality following the formula $time_{enc}(k) = 2.7 \cdot 10^{-5} \cdot k$ for $t = 16$. Using this and the formula for the number of elements in a Randomized Branching Program, we get following formula for estimating the time it takes to encode a Randomized Branching Program of length n with random matrix dimension m: $time_{enc}(n, m) = (2m + 5)^2 \cdot 1.08 \cdot 10^{-4} \cdot n^2$ Using this formula we can estimate the time it would take to encode all of the Randomized Branching Programs with matrix dimension $m = 1$.

5.1.5 Generation Analysis

The Universal Circuits that we are generating are technically not optimal, thus there is an opportunity for improvement. Nonetheless, it is important to note that the step that is by far most expensive is the generation of Branching Programs due to their inability to cope with reused gates. Even smaller Universal Circuit constructed with one of the other methods, are going to reuse gates a lot. In fact reusing gates is an inherent component of constructing Universal Circuits, because it is necessary in order to allow every possible wiring between the Universal Circuits simulation gates. This means that

s	l	Randomized Branching Programs			MJP Encoding
		length	memory	time	time
1	2	461	1200 KB	1.80 s	1125 s
2	2	53549	137 MB	179.91 s	175 d
2	3	248429	621 MB	869 s	869 s
3	2	6211757	15.5 GB	6 h	10 y
3	3	28817837	72 GB	28 h	139266 y

Table 5.2: Estimations for Randomized Branching Program generation and encoding

the Branching Program generation is always going to cause an explosion in the size and while the time cost and memory overhead of Randomizing the Branching Programs and encoding them is also not small, it is linear in the length of the Branching Program and thus less of a problem.

To summarize the cost of the construction, we can estimate a lower bound for the size in memory of a completely obfuscated Branching Program of length n as $3.5 \cdot 2500 \cdot n$ Bytes. It would take approximately $2 \cdot 10^{-6} \cdot n + 3.5 \cdot 10^{-3} \cdot n + (2m+5)^2 \cdot 1.08 \cdot 10^{-4} \cdot n^2$ seconds to compute it.

5.2 Evaluation

With this experiment we tried to understand how long the evaluation of an obfuscated circuit would take. Again we also tested the different intermediate steps to estimate which step is most expensive and thus most attractive for eventual improvements. The execution time is always averaged over all possible circuits for that circuit family and over all possible inputs to these circuits.

The estimated values for Branching Programs were calculated using the formula $2 \cdot 10^{-6} \cdot n$ which is the average time per length that was obtained from looking at the other programs. The Randomized Branching Programs are using $m = 1$ and $p = 1049$ and the estimations are obtained using the formula $4.5 \cdot 10^{-5} \cdot n$, also the average time per length from looking at the other programs.

To estimate the impact of the Multilinear Jigsaw Puzzle Encoding on the evaluation, we compared the time it took to compute the sum and the product of two elements from the plaintext space $\mathbb{Z}_p$ with the time it took to perform the same operations with encoded elements. We used different Multilinear Jigsaw Puzzles with values of $t = 16, 32, 64$ and several different values for the multilinearity k. The results (as displayed in Figure 5.3) show that the multilinearity has nearly no impact on the evaluation time, whereas using bigger values of t cause higher evaluation costs. To

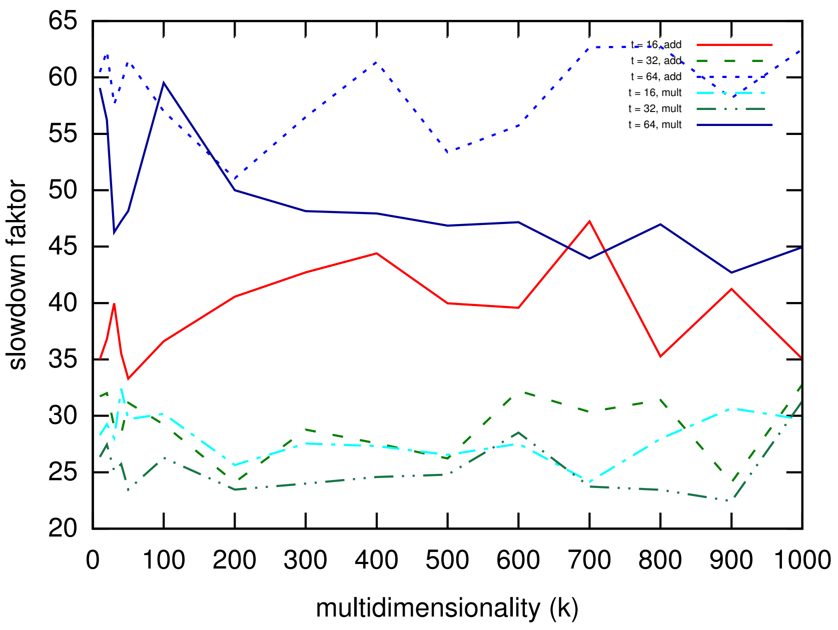

Figure 5.3: Slowdown of addition and multiplication in the encoding

estimate the cost of evaluating encoded Randomized Branching Programs, we simply took the overall average slowdown factor which was about 37. This means that we expect an accumulated cost of $1.665 \cdot 10^{-3} \cdot n$ seconds when evaluating an encoded Randomized Branching Program of length n using $m = 1$, $p = 1049$ and $t = 16$. In table 5.3 the relative slowdown factor is always in relation to the preceding construct, meaning that the Randomized Branching Program Times are compared to the Branching Program times.

s	l	circuit	Universal Circuit	Branching Program		RBP		MJP
1	2	946 μs	1010μs (1.05x)	2.24 ms	(2.21x)	27 ms	(12x)	1 s
2	2	646 μs	652 μs (1.02x)	111 ms	(170x)	2.37 s	(22x)	88 s
2	3	614 μs	537 μs (0.87x)	494 ms	(920x)	11.18 s	(22.5x)	414 s
3	2	1050μs	1140μs (1.10x)	11.6 s	(10225x)	279.52 s	(24x)	172 m
3	3	571 μs	778 μs (1.35x)	57.6 s	(74000x)	1297 s	(22.5x)	13.32 h
3	4	556 μs	808 μs (1.46x)	238.5 s	(295000x)	5366 s	(22.5x)	55 h

Table 5.3: Evaluation results

Evaluation Analysis As already mentioned above, the biggest problem of the construction are in fact the Branching Programs. This is very clearly visible, when one checks the relative costs for evaluating the different intermediate steps of the construction. Both Randomized Branching Program and the encoding with Multilinear Jigsaw Puzzles only add a linear factor to the evaluation time, the Branching Program cost on the other hand increases with the input length. This is of course partly because of our choice to set $m = 1$ and to fix t, in the 'true' construction described by the authors these values are also dependent on the Branching Program length. Nonetheless, the most important factor for the cost of evaluating the obfuscated circuits will always be the length of their corresponding Branching Programs, making the Branching Program construction the most interesting target for improving the practicality of this construction.

5.3 Fixing Universal Programs

This small experiment aims to understand the size of the final obfuscations of circuits when using the technique proposed in Section 4.6. To do so we took the generated Universal Branching Programs, fixed them for different circuits and took note of the length of the resulting Branching Programs. Notice that the structure of the index list is not affected at all when randomizing a Branching Programs or when encoding them with a Multilinear Jigsaw Puzzle, hence the memory savings will be identical (percentage wise) when one would use the same technique on a Randomized Branching Program or on an encoded Randomized Branching Program. In fact, it is to be expected that the evaluation time goes decreases by about the same percentage for all of the constructs.

s	l	length before	length after	decrease
1	2	461	224	51.4 %
2	2	53549	25984	51.5 %
2	3	248429	116928	52.9 %
3	2	6211757	3014144	51.5 %

Table 5.4: Results of Universal Circuit and Branching Program generation

6 Conclusion

This chapter contains an outline of potential improvements for the implementation and further research directions as well as the conclusion regarding the question of practicality.

6.1 Potential Improvements

Even though the paper presenting this candidate [1] was published only quite recently, there are already several works [34][35][36] improving it in different areas. Especially the improvements related to efficiency would be very interesting to examine and include in this implementation, in order to speed up the construction and hopefully make it more useful in practice.

As for optimizations of the implementation itself, there are several things that come to mind. Of course it would be better to generate the smallest Universal Circuit possible, instead of the version that is easiest to generate. Even more interesting would be a different way of generating Branching Programs which somehow softens the impact of the reuse problem. As an idea, it might be possible to directly generate Universal Branching Programs, which do not come from an application of Barrington's theorem to a Universal Circuit. In fact the Branching Programs that we generate with Barrington's theorem only use about 17 of the 120 possible permutations, which suggests that they do not utilize the full range of computational power that Branching Programs have to offer.

Additionally to these changes to parts of the algorithm, there is of course potential for optimization in the realization of the algorithm. While the matrices over $\mathbb{Z}_p$ that Sage offers are already quite optimized, it might be possible to speed up the implementation of Multilinear Jigsaw Puzzles by implementing a custom version of $Z_q[x]/(x^t+1)$ that makes use of the special modulus. Also Sage offers a lot of functionality that we do not need, it is very probable that the memory efficiency could be improved by dropping that. Another performance boost could be obtained by parallelizing the code. Most of the operations can be performed in parallel, which could speed up the evaluation considerably especially on machines with multiple cores. Finally it could bring another performance boost to move away from Python completely to a more efficient low-level language like C.

On the usability front, it would be nice and relatively straight forward to have Universal Circuits that can simulate circuits with different gates than just AND and NOT gates. For this it would be sufficient to adapt the Simulation Blocks to also include different boolean functions, like for example OR and XOR. A very interesting addition would be the ability to programmatically transform a program of some sort into a circuit, thus enabling us to obfuscate actual programs without having to design their corresponding circuits by hand. This could either be done by implementing some sort of primitive programming language for which the transformation to a circuit is trivial or maybe by leveraging already existing research in that area. For example there exist compilers [37] from C to the hardware description language VHDL [38]. While electrical circuits are different from the boolean circuits used in this thesis, it might be possible to modify them relatively easily.

One last thing that we would like to point out is the fact that it is actually possible to pre-generate large portions of the construction. Once the enormous growth has been addressed, one could generate encoded Randomized Universal Branching Programs for several input circuit families and load them as needed. This could even be outsourced to large server farms. To obfuscate a circuit a user would then only need to compute its binary description and fix the matrices of the encoded Randomized Universal Branching Program with that.

6.2 Practicality of The Candidate

The implementation presented in this thesis has several caveats, which have an impact on the performance. Most notably is the relatively primitive implementation of Universal Circuits, which creates a big overhead especially for bigger circuits. Additionally there was little optimization done in terms of algorithmic design, specialized data structures or parallelization all of which could speed up the evaluation considerably. Even though most of Sage libraries are implemented in C and optimized as far as possible, most of our code is certainly not optimized and implemented in Python, which is not necessarily the most performance oriented language.

That said, it is indisputable that the circuits that we are able to obfuscate so far are tiny and of no practical use whatsoever. Seeing that the obfuscation of a circuit with three gates and three inputs would probably take about 140 thousand years, it seems likely that even heavy optimizations of the implementation cannot overcome the inherent problems of the construction. Not to mention that this estimation is quite optimistic and more of a lower bound when one wants to implement the 'true' candidate which uses $m = 2n + 5$ and also a correct value for t.

The complete construction uses many expensive constructions and because of the way they are applied on top of each other, their overhead accumulates in a multiplicative fashion. Thus, despite the restrictions mentioned above we conclude that the construction is not ready for practical use just yet.

Bibliography

[1] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. "Candidate indistinguishability obfuscation and functional encryption for all circuits." In: *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE. 2013, pp. 40–49.

[2] *Sage*. URL: `http://www.sagemath.org/` (visited on 09/14/2014).

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. "On the (im) possibility of obfuscating programs." In: *Advances in Cryptology-CRYPTO 2001*. Springer. 2001, pp. 1–18.

[4] C. Collberg, C. Thomborson, and D. Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[5] C. Wang, J. Davidson, J. Hill, and J. Knight. "Protection of software-based survivability mechanisms." In: *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE. 2001, pp. 193–202.

[6] C. Collberg, C. Thomborson, and D. Low. "Manufacturing cheap, resilient, and stealthy opaque constructs." In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 184–196.

[7] T. Sander and C. F. Tschudin. "On software protection via function hiding." In: *Information Hiding*. Springer. 1998, pp. 111–123.

[8] Z. Vrba, P. Halvorsen, and C. Griwodz. "Program obfuscation by strong cryptography." In: *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*. IEEE. 2010, pp. 242–247.

[9] D. Low. "Java control flow obfuscation." PhD thesis. Citeseer, 1998.

[10] C. S. Collberg and C. Thomborson. "Watermarking, tamper-proofing, and obfuscation-tools for software protection." In: *Software Engineering, IEEE Transactions on* 28.8 (2002), pp. 735–746.

[11] D. Aucsmith. "Tamper resistant software: An implementation." In: *Information Hiding*. Springer. 1996, pp. 317–333.

[12] H. W. Lenstra and C. Pomerance. "A rigorous time bound for factoring integers." In: *Journal of the American Mathematical Society* 5.3 (1992), pp. 483–516.

[13] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, et al. "Factorization of a 768-bit RSA modulus." In: *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 333–350.

[14] D. Madore. *Quines (self-replicating programs)*. URL: `http://www.madore.org/~david/computers/quine.html` (visited on 09/15/2014).

[15] S. Goldwasser and G. N. Rothblum. "On best-possible obfuscation." In: *Theory of Cryptography*. Springer, 2007, pp. 194–213.

[16] A. Sahai and B. Waters. "How to Use Indistinguishability Obfuscation: Deniable Encryption, and More." In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 454.

[17] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[18] T. Holenstein. *Branching Programs and Barrington's Theorem*. URL: `http://www.complexity.ethz.ch/education/Lectures/ComplexityHS10/ScriptChapterTwelve` (visited on 09/15/2014).

[19] K. Hansen and C. Jensen. *Barrington's Theorem*. URL: `http://www.cs.au.dk/~arnsfelt/CT10/scribenotes/lecture16.pdf` (visited on 09/15/2014).

[20] J. Kilian. "Founding crytpography on oblivious transfer." In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 20–31.

[21] D. Boneh and A. Silverberg. "Applications of multilinear forms to cryptography." In: *Contemporary Mathematics* 324.1 (2003), pp. 71–90.

[22] D. Boneh. "The decision diffie-hellman problem." In: *Algorithmic number theory*. Springer, 1998, pp. 48–63.

[23] S. Garg, C. Gentry, and S. Halevi. "Candidate Multilinear Maps from Ideal Lattices." In: *Eurocrypt*. Vol. 7881. Springer. 2013, pp. 1–17.

[24] J.-S. Coron, T. Lepoint, and M. Tibouchi. "Practical multilinear maps over the integers." In: *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 476–493.

[25] M. Naor. "Bit commitment using pseudorandomness." In: *Journal of cryptology* 4.2 (1991), pp. 151–158.

[26] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping." In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM. 2012, pp. 309–325.

[27] *PyPI. the Python Package Index*. URL: `https://pypi.python.org/pypi/` (visited on 09/14/2014).

[28] L. G. Valiant. "Universal circuits (preliminary report)." In: *Proceedings of the eighth annual ACM symposium on Theory of computing*. ACM. 1976, pp. 196–203.

[29] T. Schneider. "Practical Secure Function Evaluation." MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2008.

[30] T. Schneider. *FairplayPF. Secure Evaluation of Private Functions*. URL: `http://thomaschneider.de/FairplayPF/` (visited on 09/14/2014).

[31] *NumPy*. URL: `http://www.numpy.org/` (visited on 09/14/2014).

[32] *Documentation - The Python Standard Library. time - Time access and conversions*. URL: `https://docs.python.org/2/library/time.html` (visited on 09/13/2014).

[33] S. Nilsson. *Guppy-PE. A Python Programming Environment*. 2013. URL: `http://guppy-pe.sourceforge.net/` (visited on 09/13/2014).

[34] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai. "Protecting obfuscation against algebraic attacks." In: *Advances in Cryptology–EUROCRYPT 2014*. Springer, 2014, pp. 221–238.

[35] Z. Brakerski and G. N. Rothblum. "Virtual black-box obfuscation for all circuits via generic graded encoding." In: *Theory of Cryptography*. Springer, 2014, pp. 1–25.

[36] P. Ananth, D. Gupta, Y. Ishai, and A. Sahai. "Optimizing Obfuscation: Avoiding Barrington's Theorem." In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 222.

[37] B. Buyukkurt, Z. Guo, and W. A. Najjar. "Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs." In: *Reconfigurable Computing: Architectures and Applications*. Springer, 2006, pp. 401–412.

[38] Z. Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.