In Pursuit of Clarity In Obfuscation

Prologue

The messy birth of multilinear maps in the paper "Candidate Multilinear Maps from Ideal Lattices" by Garg, Gentry, and Halevi [GGH13a] appeared to some elements of the cryptographic research community in 2012 as the fulfillment of a prophecy foretold. Surely this was at last the mathematical promised land that had been glimpsed in the advent of bilinear maps about a decade earlier, but had remained elusive in the interim. There was an energy in the air. Suddenly theoreticians were whispering amongst themselves at conferences and moving with a sense of urgency that seemed unnatural. A single word began to emerge from the whispers to become a battle cry: Obfuscation.

For unsuspecting graduate students, what then ensued was a dizzying dance of dramatic claims and relentless conference deadlines. When the dust finally settles, the human cost will perhaps best be quantified in consumed cups of coffee or extinguished whiteboard markers. The gain might ultimately be as grand as a once-in-a-lifetime breakthrough, or as modest as a pile of a little-used theorems worth hardly more than the paper they are written on. That story is yet to be fully written.

What we would like to tell here instead is a story that every scientist has lived, but only few have fully told. We want to tell the story of how we set out to solve a history-making problem, and came out of the experience not with a solution, but with a different understanding of history. We want to walk you, dear reader, through the intricate details of failure and the insights we learned along the way. But not just so you can avoid our mistakes - quite the contrary. So that when you as a scientist set out upon your own grand quest and fail time and time again along the way, you will know you have some cheerful company.

Chapter 1: The Prophecy

The desire to obfuscate programs has likely been around nearly as long as programs themselves. In the early days of computer programming, many programmers were consumed with the joy of watching their creations spring to life. But surely there were a few schemers among them who yearned to make functional but unreadable code as a way of ensuring their indispensability. The intuitive appeal of an obfuscated program becomes more primal if we think about programs as extensions of the human beings who develop them. A well-obfuscated program is a loyal servant: it does all of the tasks you delegate, while closely guarding all of your secrets. It can automatically send flowers to your lover while rebuffing the intrusions of your snooping spouse. Obfuscation represents one of the most *human* traits that programs have left to master: the ability to lie by omission.

The dream of obfuscation wafted into the cryptographic landscape alongside the discovery of public key cryptography. In the seminal paper of Diffie and Hellman [DH76], program obfuscation is suggested as a possible means to turn a private key encryption scheme into a public key encryption scheme. If we could take the encryption program of a private key scheme (which will have the secret key embedded in it) and sufficiently mask its inner workings so that it does not reveal unnecessary information about the key against a resource-bound attacker, then we could safely publish this obfuscated encryption program as a public key. But the concrete public key encryption scheme proposed by Diffie and Hellman did not work this way. They left it as an open question whether such an approach could be made to work. This thought dangled before the minds of cryptographers for decades, like a lure on a dog racing track.

Next in the old testament of obfuscation, there came some bad news. A formal study of program obfuscation, undertaken by Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, and Yang in 2001 [BGI+01], established that arguably the most natural formulation of the problem is impossible to solve. To understand the crux of their result, we must ponder how to rigorously define what secure program obfuscation should mean. Intuitively, a well-obfuscated piece of code should leak no more information than the bare minimum it must need to perform its function. It is tempting to say that having access to obfuscated code that computes a function f should be like having access to f "in a black box." This is like a magic 8-ball notion of security: you imagine that you can ask the magic black box for $f(x)$ on any particular input x of your choosing, and it spits out $y = f(x)$ seemingly out of thin air. There are no intermediary steps of the computation that you can observe, and no differences in the running time, or energy consumption, etc.

Of course, this impenetrable black box is a fantasy. We know real code does not behave like this. Nonetheless, we might hope to achieve what is called "Virtual Black Box Obfuscation," meaning that a person who is handed the actual obfuscated code cannot manage to do anything with it that they could not also do if given this magical black box instead. If we can achieve this for a particular function f, then it is true that the real code is not really a black box, but it effectively behaves like one.

The Barak et al. paper showed that this cannot comprehensively be achieved. In retrospect, their counterexample is reasonably intuitive. The thing you can always do with code that computes a function f and sometimes cannot do with a black box for f is ... drum roll please... produce code that computes the function f. This is a bit more subtle than it seems, so let's walk through this slowly.

For some functions f, it may be easy to derive code for computing f from access to a black box version. A basic example is a function with a very small input space. If f is a function from two bits to 1 bit, for example, like XOR, than you can simply make an exhaustive table that tells you exactly what f outputs for each possible input:

x	y	x $\otimes$ y
0	0	0
0	1	1
1	0	1
1	1	0

This lookup table can serve as "code that computes the function $f := \otimes$", and it satisfies this very strong notion of virtual black box obfuscation. The table obviously tells you no more about f than you can learn by asking a black box, because all you have to do is ask the box for the outputs on each of the 4 possible input pairs. As input spaces get larger, however, a look up table that simply lists the function output for each input diverges farther and farther from our internalized notion of what constitutes "code." A table that is too large to be stored in any plausible amount of memory does not really pass muster if we want a *usable* representation of a function f. For this reason, we tend to limit our selves to functions and code that can be represented in size *polynomial* in the input length. That is to say, if we allow inputs that are n-bit strings, we want code that can be stored and computed in time $poly(n)$, not 2^n, etc.

Similarly, when we think about querying a black box that computes a function, we will limit ourselves to *poly*(n) queries.

For concreteness, let's consider a fixed input length of $n = 512$ bits. We will define a function $C_{\alpha,\beta} : \{0,1\}^n \rightarrow \{0,1\}^n$ that takes in an input x of length 512 and outputs a value y that is also of length 512. The behavior of $C_{\alpha,\beta}$ will be determined by α and β, which will be hard-coded constants also of length 512. The code of $C_{\alpha,\beta}$ will simply check if the input $x = \alpha$. If so, it output β. If not, it will output a string of 0's. In other words, pseudocode for $C_{\alpha,\beta}$ might look like:

```
if $x = \alpha$, output $\beta$
else, output $0^n$
```

In this form, it is clear that the code gives away more than a black box for computing $C_{\alpha,\beta}$ would, since we limit ourselves to querying the box at most *poly*(n) times. There are 2^n possible values of x, so if α is chosen uniformly at random and unknown to us, we are highly unlikely to discover if by querying *poly*(n) values for x. Thus, the black box version of $C_{\alpha,\beta}$ should effectively hide α (and β as well), while this pseudocode version reveals both.

It may seem hard to imagine how *any* piece of code that performs a check like "if $x = \alpha$" could possibly avoid revealing α. But actually we can hope to do this part with fairly standard cryptographic tools. If we choose a suitable hash function H, we can replace a check like "if $x = \alpha$" with "if $H(x) = H(\alpha)$". Now the hard-coded constant in the code is $H(\alpha)$, not α itself. So if H is a one-way function (meaning that it is hard to find α given only $H(\alpha)$, then our code can effectively hide α despite performing this check.

The problem appears as we try to get fancier. Let's define another function $D_{\alpha,\beta}$ that takes in a piece of code and tests whether or not that code computes the function $C_{\alpha,\beta}$. At this point, you should object - how could such a $D_{\alpha,\beta}$ be defined compactly? Won't it need to exhaustively test the input code on all possible values? And what if the input code doesn't always terminate? Well, let's cut ourselves some slack. Let's imagine that $D_{\alpha,\beta}$ merely runs the input code on α and a polynomial number of other random inputs, each for a polynomial number of steps. If the input code spits out β on input α in that amount of time, and all other tested inputs spit out 0^n in that amount of time, then $D_{\alpha,\beta}$ outputs 1. Otherwise, $D_{\alpha,\beta}$ outputs 0.

Now let's imagine what would happen if we could obtain obfuscated code that computes $C_{\alpha,\beta}$, which we'll denote by $\mathcal{O}(C_{\alpha,\beta})$, and obfuscated code that computes $D_{\alpha,\beta}$, which we'll denote by $\mathcal{O}(D_{\alpha,\beta})$. Let's continue to suppose that α, β are uniformly random from $\{0,1\}^n$ and unknown to us. We can run $\mathcal{O}(D_{\alpha,\beta})$ on $\mathcal{O}(C_{\alpha,\beta})$ and get an output of 1, since the obfuscated code for $C_{\alpha,\beta}$ should still behave as $C_{\alpha,\beta}$ does, and hence should pass the tests performed by $\mathcal{O}(D_{\alpha,\beta})$. However, if we were instead given two black boxes, one that computed $C_{\alpha,\beta}$ and one that computed $D_{\alpha,\beta}$, and were only able to query them for a *poly*(n) number of times, it would be nearly impossible for us to discover a way to make either box output 1!

This is of course the fancier way of saying that what one cannot do without code is produce code. At least for classes of functions that include these kind of $C_{\alpha,\beta}, D_{\alpha,\beta}$ examples. Faced with this disappointing truth, cryptographers exhibited each of the 5 stages of grief:

Denial Well, maybe those commercial products for program obfuscation work in practice. Let me try to break one.

ten minutes later...

Oh, nevermind.

Anger This counterexample is stupid pants.

Bargaining Ok, we can't have virtual black box obfuscation. But maybe we can have something weaker? Dark Grey box obfuscation? Heather Gray box obfuscation? Pewter box obfuscation? Indistinguishability obfuscation[1]? Come on, *please*?

Depression I'm never going to get tenure from studying this at this rate.

Acceptance Forget obfuscation. Let's go back to working on multi-authority, post-quantum, identity-based, partially homomorphic, aggregate signatures. Someone will someday need them!

And that's where the field stood, for over a decade. Until an obfuscated God appeared before three wise men, Garg, Gentry, and Halevi, and told them to bring forth an age of multilinear maps.

Chapter 2: The Flood

To understand multilinear maps and what they have to do with obfuscation, it's best to start with discrete log basics. We will let G denote a group of prime order p, generated by g. We typically assume that the group G has an efficiently computable group operation and an efficiently computable "zero test," meaning that even if there are multiple ways to represent the same element of G, we can recognize when two representations correspond to the same element. This is equivalent to being able to recognize any suitable representation of the identity element, g^0.

The discrete logarithm problem can then be described as follows: we sample an exponent a uniformly at random from $\mathbb{Z}_p$, and we compute g^a. Given g and g^a, the discrete logarithm problem requires that we compute a. Clearly this can be done if given enough time, as $\mathbb{Z}_p$ is finite, and we could simply try all possibilities. But if p is sufficiently large, this brute-force approach is not practical. Naturally, there are better approaches to computing a, some of which work only for certain groups G. At this point, it is still strongly believed that there are good choices for G which make computing g^a from g and a easy but computing a from g and g^a quite hard. Hard enough that it can be assumed to be impractical.

We will assume that G is such a group. Then if we are given g^a and g^b, we can easily compute g^{a+b}, but it still is hard to compute a, b, or $a+b$, at least when a and b are randomly chosen. In some sense, this provides a bit of obfuscation: we know that we are adding in the exponent, but we don't know what the values are. This is kind of like hiding the hard-coded constants in a piece of code, though we reveal what operation we are performing on them.

But to be useful, even obfuscated code should spit out the end result when it is done computing. When we obtain g^{a+b}, the result $a+b$ is still stuck in the exponent, and in general it is hard to get it out. There are a few exceptions to this though - if we know that an exponent is within a pre-determined polynomial range, for instance, we can find it exhaustively. And we can always determine some very basic information about the exponent, like whether it is zero or nonzero as an element of $\mathbb{Z}_p$.

This leads us to a potential way to obfuscate a very simple function: namely the characteristic function of a point. We will let P_α denote the function from $\mathbb{Z}_p$ to $\{0,1\}$ that is equal to 1 on $\alpha \in \mathbb{Z}_p$ and equal to 0 on all other inputs. If we provide the values g and g^α, then the function P_α can be computed on an arbitrary input $x \in \mathbb{Z}_p$ by computing g^x and comparing this to g^α. If the group G's zero test tells us that these elements are the same, then $P(x) = 1$. Otherwise, $P(x) = 0$.

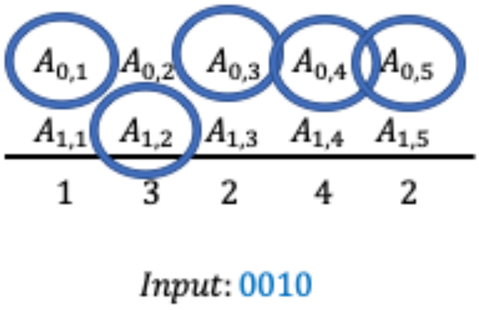
[1] This one is real, as we shall see.

The discrete log assumption then asserts that if α is randomly chosen, we have obfuscated our function P_α at least to the extent that the hard-coded value α cannot be efficiently extracted from the code. It is tempting to see how far we can push this kind of trick. We'd like to be able to perform more complex computations in the exponent of a group in such a way that only the final function result can be efficiently extracted. For boolean functions that always output 0 or 1, we could imagine trying to arrange it so that 0 corresponds to a final result of g^0, while 1 corresponds to a final result of g^c for some nonzero value c. If all of the intermediary exponents we obtain throughout the computation stay away from easily recognizable values like 0, we might hope we have achieved some meaningful level of obfuscation.

Let's start by trying to put more interesting boolean functions into a form that may be compatible with computation in an exponent. One possible template for this is matrix branching programs. A matrix branching program represents a boolean function $f : \{0,1\}^n \to \{0,1\}$ through pairs of matrices. We will refer to each pair of matrices (denoted $A_{0,i}$ and $A_{1,i}$) as a "slot," and each slot is associated with an index between 1 and n. For example, a matrix branching program with $n = 4$ and 5 slots may look something like this:

$$\begin{array}{ccccc} A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} & A_{0,5} \\ A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} \\ \hline 1 & 3 & 2 & 4 & 2 \end{array}$$

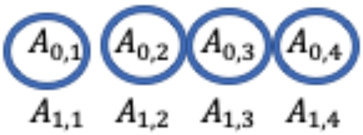
To avoid overburdening ourselves with notation, we have somewhat casually written the associated indices in $[n] = [4]$ below each slot. To evaluate the function that this matrix branching program represents on an input, simply select $A_{0,i}$ for any slots i where the associated input bits are equal to 0, and select $A_{1,i}$ for any slots i where the associated input bits are equal to 1. Then take the product of all these matrices in order, and see if the result is the identity matrix or not. For example, if our input is 0010 for the above matrix branching program, we would take the product $A_{0,1}A_{1,2}A_{0,3}A_{0,4}A_{0,5}$ and compare it to the identity matrix. All of our matrix entries and our computations will be in $\mathbb{Z}_p$. We can visualize this selection of matrices as:



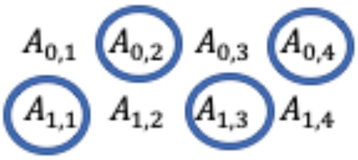
Note that the input bits associated to the slots do no have to go in order: in this case, the second slot is associated with the third input bit, which is the only bit set = 1 for our particular input.

Naturally, the matrix dimensions must be compatible for this product to make sense. We can restrict ourselves to considering square matrices of a fixed common size, for instance, to keep things simple. In fact, it's typical to make the matrices each 5×5.

Let's see what the characteristic function of a point might look like in this template. If we want to represent the function $f : \{0,1\}^4 \to \{0,1\}$ that equals 1 only on the input string 0000, we could choose the matrices $A_{0,1}, A_{1,1}, A_{0,2}, A_{1,2}, A_{0,3}, A_{1,3}, A_{1,4}$ all uniformly at random from $\mathbb{Z}_p^{5\times 5}$, and then set $A_{0,4} := (A_{0,1}A_{0,2}A_{0,3})^{-1}$. In this case, slot i would correspond to input bit i, so the evaluation for the input 0000 would compute the product $A_{0,1}A_{0,2}A_{0,3}A_{0,4} = I$. With high probability, each of our randomly chosen matrices will be invertible, so this computation will succeed. Also, if p is reasonably large, then the other matrix products which correspond to other inputs will not equal the identity matrix, except with negligible probability. In cases where the i^{th} slot corresponds to the i^{th} input bit, we can omit the extra indices from our visualization:



Input: 0000



Input: 1010

It turns out we can place even more stringent requirements on the matrices if we want to, and still manage to represent an impressive variety of functions. Even if we insist that all of our matrices be 5×5 permutation matrices, and all of the products that correspond to valid input evaluations result in either the identity matrix I or some other fixed matrix B, we can still represent all boolean circuits this way! This is asserted by Barrington's theorem, which provides a construction that uses a number of slots that is exponential in the depth of the boolean circuit computing the desired function f. So if we restrict ourselves to circuits of logarithmic depth in our security parameter, this can qualify as being polynomial time.

Now let's revisit our obfuscation of a point function $P_\alpha(x) = 0$ for all $x \neq \alpha$ and $P_\alpha(\alpha) = 1$ for $\alpha \in \mathbb{Z}_p$. We obfuscated the value of α by putting it in the exponent of a group G of prime order p, and gave out g, g^α instead of α itself. The function can still be evaluated on an input $x \in \mathbb{Z}_p$ by computing g^x and comparing it to g^α: equivalently, computing $g^{x-\alpha}$ and testing if it is the identity element g^0. But if we only have the group operation on G and nothing else to work with, we can't do anything much fancier in the exponent efficiently: only linear computations.

It is natural to want to combine the magical hiding properties of putting something "in the exponent" of a group where the discrete logarithm is hard with fancier computation. The concept of a multilinear map arises from this dream. A k-linear map is something that takes in k different group elements, like g^{a_1}, g^{a_2}, $\ldots$, g^{a_k} and produces an element in a new group (called the target group) whose exponent is the k-way product $a_1 \cdots a_k$. If the target group does not itself have any efficient n-linear maps for $n > 1$, then we can think of this as a way of enforcing that k-way products can be computed in the exponent, but not any higher degree combinations. So if we have a 2-linear map, we can essentially perform 1 matrix multiplication in the exponent. If we have a 3-linear map, we can perform two successive matrix multiplications in the exponent. If we have a k-linear map, we can perform $k-1$ successive matrix multiplications in the exponent, and then no more.

To get some useful information out after we have applied the multilinear map, we will assume that the target group has an efficient "zero test," that allows us to test if the resulting exponent in the target group is 0 or not. So if we have a matrix branching program with k slots, we can hope to somewhat hide these matrices by putting them in an exponent of a k-linear group and still evaluate the function efficiently and correctly by zero testing a certain coordinate of the final product matrix, a coordinate where the identity matrix I has a 0 but the other possible output matrix B does not.

Things are not quite that easy - there needs to be some additional randomization of the matrices in the branching program, for example, as they may have some tell-tale zero entries themselves, or other obvious correlations. And of course, proving things formally when dealing with unruly k-linear polynomials in the exponent is a bit of a pain. But it is not that surprising that this approach does ultimately work - if we have an efficient k-linear map into a target group with an efficient zero-test and not much other structure (e.g. [GGH$^+$13b, BGK$^+$14, GLSW15] and many other papers).

Good candidates for 2-linear maps (a.k.a. bilinear maps) have been known for some time, arising from the more classical structures of elliptic curve cryptography. But even 3-linear maps that could plausibly have the right cryptographic properties remained a distant dream, until the work of [GGH13a] opened the flood gates. Like Noah gathering the animals, the cryptography research community responded by rounding up the applications k by k, redesigning old primitives by the new methods, constructing new primitives by the new methods, and predictably enshrining the new methods as a purpose onto themselves. It was truly an exhilarating time to be a cryptographer! And then the ark sprung a leak...

Chapter 3: The Antiheros

The original construction of multilinear maps in [GGH13a] has not been decisively broken, but it has not been free from unpleasant surprises either. A string of partial attacks (e.g. [CHL$^+$15, CGH$^+$15, CLLT16, MSZ16, CLLT17]) on [GGH13a] and its descendants (e.g.[CLT13]) have severely shifted the boundary of what we know about the security of these constructions, with many hoped-for modes of usage now landing on the insecure side. As of this writing, this ground remains stubbornly shaky. There are no widely revered and long-standing constructions, nor is there a killer attack that obliterates all hope for this line of work. Perhaps time will yield a more satisfying state. There is much reason to remain optimistic: after all, the notion of a cryptographically useful 3-linear map does not seem so fundamentally different from a cryptographically useful bilinear map, and in the bilinear setting we have seemingly reached much firmer ground.

Before Gentry's breakthrough in fully homomorphic encryption in 2009 [Gen09], one may have justifiably felt similar doubt about the concept of performing accurate computations on top of encrypted data. That line of work has now matured splendidly (at least from a theoretician's perspective), and has yielded progressively simpler constructions from progressively firmer mathematical foundations (e.g. [GSW13]). Multilinear maps even feel tantalizingly close to fully homomorphic encryption: performing a k-wise multiplication in an "exponent" is perhaps not so different philosophically from multiplying k underlying plaintexts without decrypting. But the difference comes when one wants to get an answer out - in fully homomorphic encryption, it is intended that the final result of the computation remains obscured by the encryption, seemingly impenetrable to someone who does not have the secret key. But in k-linear maps (or obfuscation, for that matter), a zero test on the final output needs to be efficiently performed without comprising the rest of the computation. So additional information must be given out to perform this test, and unsurprisingly it is this additional information that provides the fuel for the non-trivial attacks on current multilinear map constructions.

Still, this may turn out to be a technical, rather than fundamental problem. The people who decisively solve this problem will likely be heralded as heros in the view of future cryptographic historians.[2] But we are here to chronicle a perhaps less heroic pursuit - the pursuit of an answer to the question: how much of this multilinear map stuff is really necessary for obfuscation in the first place?

It was several years ago now that we set out to explore this question. We started by asking: what kinds of damaging, unnecessary information do matrix branching programs really leak? We felt this was a natural starting point, since if one is going to avoid the temptation of using a multilinear map and hoping it hides "everything" except the zero test result, then one should first impose some discipline by trying to identify what really needs to be hidden.

Let's go back to considering a matrix branching program in the form:

$$\begin{array}{ccccc} A_{0,1} & A_{0,2} & A_{0,3} & A_{0,4} & A_{0,5} \\ A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} \\ \hline 1 & 3 & 2 & 4 & 2 \end{array}$$

We may first observe that the function we are representing is ultimately only affected by certain products of these matrices, such as $A_{0,1}A_{0,2}A_{1,3}A_{0,4}A_{1,5}$, which corresponds to the function evaluation on the input string 0100. This means we can add some more randomness to the individual matrices themselves, as long as we do not disturb these meaningful products. Let's sample some random invertible 5×5 matrices $R_1, \ldots, R_4$ over $\mathbb{Z}_p$ and sprinkle them into our matrix branching program like so:

$$\begin{array}{ccccc} A_{0,1}R_1 & R_1^{-1}A_{0,2}R_2 & R_2^{-1}A_{0,3}R_3 & R_3^{-1}A_{0,4}R_4 & R_4^{-1}A_{0,5} \\ A_{1,1}R_1 & R_1^{-1}A_{1,2}R_2 & R_2^{-1}A_{1,3}R_3 & R_3^{-1}A_{1,4}R_4 & R_4^{-1}A_{1,5} \\ \hline 1 & 3 & 2 & 4 & 2 \end{array}$$

This new matrix branching program computes the same function as before, since taking any 5-way product that corresponds to a legitimate evaluation yields the same result, e.g.

$$A_{0,1}R_1R_1^{-1}A_{0,2}R_2R_2^{-1}A_{1,3}R_3R_3^{-1}A_{0,4}R_4R_4^{-1}A_{1,5} = A_{0,1}A_{0,2}A_{1,3}A_{0,4}A_{1,5}.$$

However, these additional matrices $R_1, \ldots, R_4$ can help destroy unnecessary structure. For instance, it may be the case that $A_{1,2} = A_{1,3}$, but nonetheless $R_1^{-1}A_{1,2}R_2 \neq R_2^{-1}A_{1,3}R_3$, at least with high probability over the choices of R_1, R_2, R_3.

Kilian [Kil88] observed that in fact, if we take only one matrix from each "slot," the joint distribution is now only determined by the corresponding 5-way product. For example, the joint distribution of the 5 matrices

$$A_{0,1}R_1, \;\; R_1^{-1}A_{0,2}R_2, \;\; R_2^{-1}A_{1,3}R_3, \;\; R_3^{-1}A_{0,4}R_4, \;\; R_4^{-1}A_{1,5}$$

can be sampled by knowing only the product $A_{0,1}A_{0,2}A_{1,3}A_{0,4}A_{1,5}$, and is independent of any further details of the individual matrices $A_{0,1}$, $A_{0,2}$, $A_{1,3}$, $A_{0,4}$ and $A_{1,5}$.

To see why this is true, suppose there are any two sets of five square invertible matrices that have the same product: $ABCDE = FGHIJ$. We can define R_1 such that $AR_1 = F$, simply by setting $R_1 = A^{-1}F$. We can define R_2 such that $R_1^{-1}BR_2 = G$ simply setting $R_2 = B^{-1}R_1G$.

[2] In an imaginary future where there are cryptographic historians.

Similarly we can set $R_3 = C^{-1}R_2H$ and then $R_4 = D^{-1}R_3I$ to ensure that $R_2^{-1}CR_3 = H$ and $R_3^{-1}DR_4 = I$. Finally, we now must have that

$$\begin{aligned} R_4^{-1}E &= I^{-1}R_3^{-1}DE \\ &= I^{-1}H^{-1}R_2^{-1}CDE \\ &= I^{-1}H^{-1}G^{-1}R_1^{-1}BCDE \\ &= I^{-1}H^{-1}G^{-1}F^{-1}ABCDE \\ &= J, \end{aligned}$$

since we began by assuming that $ABCDE = FGHIJ$. Thus, there is always precisely one setting for the invertible matrices $R_1, \ldots, R_4$ that will transform A, B, C, D, E into F, G, H, I, J when used as randomization in this way, while still preserving the 5-way product. Hence, if we choose these matrices $R_1, \ldots, R_4$ uniformly at random under these constraints, we will produce a distribution for $AR_1, \ldots, R_4^{-1}E$ that depends only on the product $ABCDE$.

This implies that if we only produce the matrices corresponding to the evaluation of our matrix branching program *on a single input*, we can achieve an obfuscation property information-theoretically: our output still only depends upon the output of the function on this point. But this alone is not very useful in the obfuscation setting: clearly we want to be able to evaluate the function on many more points.

So what information beyond the function values gets compromised when we give out *all* of the matrices? Potentially quite a lot. We can, for example, see whether $A_{0,2} = A_{1,2}$, since multiplying each of these by the *same* invertible matrices R_1^{-1} and R_2 on the left and right respectively will not obscure such an equality.

This might be important, for example, if the input bit that is referenced in deciding which of $A_{0,2}$ and $A_{1,2}$ to throw into the computation is not referenced anywhere else. Under such circumstances, $A_{0,2} = A_{1,2}$ implies that the value of this bit never influences the output of the function, a fact that may be difficult to discern from only oracle access to the function.

To make this even more concrete, let's tweak our example of a point function just a tiny bit. Consider a function $f : \{0,1\}^4 \to \{0,1\}$ that is equal to 1 on two input strings: 0000 and 0100. Alternatively, we could express the support of f with the notation $0 * 00$, indicating that the first, third, and fourth bits must be 0, but the second bit value is a "wildcard" that can be either 0 or 1. This function f can be computed with a matrix branching program of length 4 where the i^{th} matrix pair corresponds to the i^{th} input bit by choosing $A_{0,1}, A_{1,1}, A_{0,2}, A_{0,3}, A_{1,3}, A_{1,4}$ all uniformly at random from $\mathbb{Z}_p^{5\times5}$, and then setting $A_{1,2} = A_{0,2}$ and $A_{0,4} = (A_{0,1}A_{0,2}A_{0,3}^{-1})$. With high probability, only the two matrix products $A_{0,1}A_{0,2}A_{0,3}A_{0,4}$ and $A_{0,1}A_{1,2}A_{0,3}A_{0,4}$ will be equal to the identity matrix, as desired.

Now with only 4 input bits, all of the behavior of the function can be succinctly learned by evaluating the function on all $2^4 = 16$ possible inputs, but you can easily imagine how this generalizes to n input bits. If we keep the support description as $0 * 00...$ and allow the number of $0's$ to grow, it doesn't take long before it becomes impractical to evaluate the function on all inputs. If we pick a function with a more arbitrary but similarly structured support, like 00110101011∗010100001..., you can start to see that it could be hard to find which bit position is the wildcard if you only get oracle access to the function. In order to determine that bit position i is the wildcard by querying the function values, you have to find a input in the support of the function, toggle bit position i, and see that it stays in the support. This can be done by process of elimination on the bit positions *after* you have found an input in the support, but it's not clear how to get this starting point in polynomial number of queries when the pattern required for the other bits is unpredictable.

In contrast, if you are given a matrix branching program where $A_{0,i} = A_{1,i}$ for only the wildcard position i, determining the wildcard position becomes easy! As we have already noted, the Kilian trick of multiplying by additional random matrices that will be applied in the same way to $A_{0,i}$ and $A_{1,i}$ does not solve this problem.

It is tempting to try something like the following: let's take each of our 5×5 matrices $A_{b,j}$ and embed it as the bottom right quadrant of a larger 10×10 matrix that we will call $\tilde{A}_{b,j}$. In the top left quadrant, we will put a random diagonal matrix $D_{b,j}$ in $\mathbb{Z}_p^{5\times 5}$. In the top right and bottom left quadrants (the off-diagonal quadrants), we will put all $0's$. Note that the random top left block we choose for $\tilde{A}_{0,j}$ will be different than the random top left block we choose for $\tilde{A}_{1,j}$, *even if* $A_{0,j} = A_{1,j}$:

$$\tilde{A}_{b,j} = \left[\begin{array}{c|c} D_{b,j} & 0 \\ \hline 0 & A_{b,j} \end{array} \right]$$

Now let's further randomize the matrices as $R_{j-1}^{-1}\tilde{A}_{b,j}R_j$, for j from 1 to n. Note that this time we've added "bookends" R_0 and R_n, so the final matrix product corresponding to an input with bit values b_i will be:

$$Y := R_0^{-1} \left(\prod_{i=1}^{n} \tilde{A}_{b_i,i} \right) R_n.$$

Note that this seems to obscure equalities of the underlying matrices $A_{0,j}, A_{1,j}$ for any value of j. Our bookends further prevent the trivial extraction of structure such as $A_{0,1} = A_{1,1}$ or $A_{0,n} = A_{1,n}$ by leveraging the block structure of the $\tilde{A}_{b,1}$ and $\tilde{A}_{b,n}$. But, you may object, how do we evaluate the function? Don't the R_0 and R_n matrices also obscure *the output*?

Let's try to correct for that. Consider the structure of $\prod_{i=1}^{n} \tilde{A}_{b_i,i}$: this will have $0's$ in the off-diagonal quadrants, a diagonal matrix in the top left quadrant, and $\prod_{i=1}^{n} A_{b_j,j}$ in the bottom right quadrant. If this is an input that evaluates to 1, then $\prod_{i=1}^{n} A_{b_j,j}$ will be the 5×5 identity matrix I. Let's consider a vector $v \in \mathbb{Z}_p^5$, chosen uniformly at random. We'll let $\tilde{v}$ denote the column vector in $\mathbb{Z}_p^{10}$ that is formed by putting 0's in the first 5 entries and using v as the last 5 entries. Now let v' denote the column vector $R_n^{-1}\tilde{v}$ and x denote a row vector such that column vector $xR_0^{-1}\tilde{v} \equiv 0 \bmod p$. If $\prod_{i=1}^{n} A_{b_j,j} = I$, we will have $xYv' \equiv 0 \bmod p$. If $\prod_{i=1}^{n} A_{b_j,j} \neq I$, and further maps $\tilde{v}$ to something *not* orthogonal to R_0^{-1} (which is easy enough to arrange), then xYv' will *not* be congruent to 0 when the input does not evaluate to 1. Hence we can give out x and v' as additional tools to enable an evaluator to compute the underlying function, and this does not look like it destroys our progress in obscuring equalities among the underlying matrices $A_{b,i}$ for a wildcard position i.

But alas, looks can be deceiving. Let's reexamine the two matrices that now correspond to a wildcard position i: $R_{i-1}^{-1}\tilde{A}_{0,i}R_i$ and $R_{i-1}^{-1}\tilde{A}_{1,i}R_i$. Suppose we invert this first matrix and multiply the result on the left of the second, obtaining

$$Z := R_i^{-1}\tilde{A}_{0,i}^{-1}\tilde{A}_{1,i}R_i.$$

Since $A_{0,i} = A_{1,i}$, we have that $\tilde{A}_{0,i}^{-1}\tilde{A}_{1,i}$ is a diagonal matrix, with the last 5 diagonal entries equal to 1. This means it has a 5-dimensional eigenspace with eigenvalue 1 (something unlikely to happen when i is not a wildcard position). We observe that the matrix Z is *similar* to $\tilde{A}_{0,i}^{-1}\tilde{A}_{1,i}$, which means it will have the same eigenvalues, and the same dimensions for the corresponding eigenspaces. Hence by looking at the eigenvalues of Z, we can identify i as a wildcard position!

We may be tempted to keep going further down this path, sprinkling more randomness here and there, adding dummy dimensions that may balance out asymmetries in eigenvalues, etc., but it seems very difficult to define success and know when to stop. Instead we will take a step back, and rethink our approach.

Chapter 4: Hidden Subgroups and The Inter-Column Shell Game

All of what we have learned so far points us to an intuitive and perhaps underwhelming conclusion: we need *something* like multilinear maps to computationally hide *something* about our matrices in order to have a promising path towards obfuscation of matrix branching programs. We have seen it is necessary to hide matrix equalities. We have further seen that it is necessary to hide eigenspace structures. But how much hiding is enough? If we manage to avoid the eigenspace attack we just detailed, are we safe? How could we know?

We can approach these questions from a complementary perspective: what is it *sufficient* to hide? One answer to this can be found in [GLSW15], where it is shown that a property called "intercolumn security" can be leveraged to prove indistinguishability obfuscation for matrix branching programs. We could quickly state a technical definition of intercolumn security and a technical definition of indistinguishability obfuscation, and rush forward with explaining how intercolumn security can form the backbone of a proof of indistinguishability obfuscation, but this would probably make intercolumn security and indistinguishability obfuscation feel like random and magical objects, and would not tell the more rambling human story of how they came to be formulated and related in the first place. So be patient - as often in our quest, we will take the scenic route.

The motivation for intercolumn security arises in the search to find a happy medium between security properties that are very ambitious and hence easily suffice for obfuscation but seemingly difficult to instantiate, and security properties that seem like reasonable targets for imperfect but non-trivial constructions of multilinear maps to achieve. For the multilinear map candidates we have so far, proving concrete and satisfying security properties from time-tested mathematical foundations remains a challenge. So much of our intuition for what *might* be achievable security properties comes from extrapolating our knowledge of bilinear maps, for which we have more established candidates.

This is problematic, because the candidate multilinear maps we have are not made from the same mathematical building blocks as the pre-existing bilinear maps. However, we will try to stick to relatively simple and abstract properties that are not too deeply intertwined with any particular approach for implementing multilinear maps, and hence we may be optimistic that such structures could be instantiated in a richer variety of ways in the future.

One particularly helpful abstract structure in cryptographic bilinear groups is the structure of subgroups. This most naturally arises when the group order is a composite number N that is the product of a few large primes. There are many methods for building subgroup-style structures in prime order groups (e.g. [OT10, Lew12, CM14, CGW15]), but will describe things here in the composite order setting for simplicity. The most basic example is $N = pq$, where p and q are distinct large primes. Upon encountering such a group, a mathematician will say: "Ah, a cyclic group of order $N = pq$. This is isomorphic to $\mathbb{Z}_p \times \mathbb{Z}_q$." The mathematician is usually fully satisfied by this, and will be surprised by any follow-up questions.

It is certainly true that for those with a good understanding of the structure of $\mathbb{Z}_p$ and $\mathbb{Z}_q$ individually, the structure of $\mathbb{Z}_p \times \mathbb{Z}_q$ is an intuitive consequence. Every element of $\mathbb{Z}_p \times \mathbb{Z}_q$ has a "p-part" and a "q-part," and these pieces behave independently in parallel, like a teen pair of twins who insist on dressing in vastly different styles as they are stuck going to school together. Formally, we can express each element of our group that is isomorphic to $\mathbb{Z}_p \times \mathbb{Z}_q$ as $g_p^r g_q^t$, where g_p is a generator for its order p subgroup, g_q is a generator for its order q subgroup, r is an element of $\mathbb{Z}_p$, and t is an element of $\mathbb{Z}_q$. The p and q parts are stuck together as part of the same group element, but their independence is established by the Chinese Remainder Theorem, which says that if an element of $\mathbb{Z}_N$ is chosen uniformly at random, the induced distributions on its modulo p reduction and its modulo q reduction are independent. This means that if we choose a number uniformly at random between 0 and $N-1$, and then let r denote its reduction modulo p and t denote its reduction modulo q, revealing r gives away *no information* about t: the conditional distribution of t for any given value of r is still uniformly random over $\mathbb{Z}_q$.

All of this is so intuitive to seasoned mathematicians that one math graduate student, upon attending an elliptic curve cryptography talk for the first time, asked in confused shock: "but what about these groups could possibly be hard?"

Well, my mathematician readers, what's most interesting here for cryptographic purposes is the fact that the isomorphism from our group to $\mathbb{Z}_p \times \mathbb{Z}_q$ *may not be efficiently computable.* The fact that this underlying, decomposable structure exists does not mean that we can efficiently compute discrete logs, nor does it mean that when faced with an arbitrary representation of the element $g_p^r g_q^t$, we can efficiently separate out its p and q parts. Crucially, this last part relies on the assumption that factoring N into p and q is hard. If we know the individual values of p and q, and the group operation itself is efficiently computable, we can take $g_p^r g_q^t$ and raise it to the p, thereby killing its p-part and obtaining g_q^{tp}, which we can raise to the inverse of p modulo q to obtain the original q-part, g_q^t. But if we don't know the factorization of N, its not clear how to separate out the p and q parts, and this is the backbone of a commonly used computational hardness assumption, known as the subgroup decision assumption.

The basic subgroup decision assumption for such a group of order $N = pq$ states that given elements $g_p, g_p^r g_q^t, T$ where r, t are chosen uniformly at random and T is either a random element of the whole group or a random element of the subgroup of order p, it is computationally hard to tell which. Note that we have *not* given out g_q, or any other element promised to be solely in the subgroup of order q. This is crucial in a bilinear group, where we have an efficiently computable bilinear map $e : G \times G \to G_T$ that maps pairs of group elements into a new "target group" G_T, essentially performing a one-time multiplication in the exponent:

$$e(g^a, g^b) = e(g, g)^{ab}$$

We can write this out more explicitly in terms of the p and q parts, using $g = g_p g_q$:

$$e(g_p^a g_q^a, g_p^b g_q^b) = e(g_p g_q, g_p g_q)^{ab}.$$

Don't let the symmetry of the notation here mislead you: we can think of a as a modulo N number, but the property of a that matters for the value of g_p^a is the reduction of a modulo p, while the property of a that matters for the value of g_q^a is the reduction of a modulo q. And for a randomly chosen a, these are uncorrelated.

It is a further consequence of bilinearity that we can write:

$$e(g_p g_q, g_p g_q) = e(g_p, g_p) e(g_p, g_q) e(g_q, g_p) e(g_q, g_q).$$

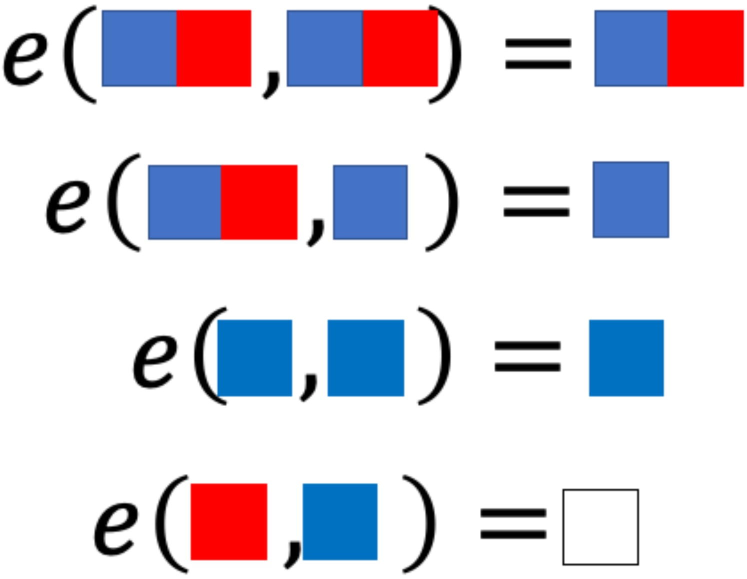
Now let's contemplate one of these middle terms, $e(g_p, g_q)$. Since p and q are distinct primes, there exists a value c modulo N such that $c \equiv 1 \bmod p$ and $c \equiv 0 \bmod q$. Thus, $g_p^c = g_p$, and furthermore,

$$e(g_p, g_q) = e(g_p^c, g_q) = e(g_p, g_q)^c = e(g_p, g_q^c) = e(g_p, g_q^0) = 1,$$

where 1 here represents the multiplicative identity element in G_T. The term $e(g_q, g_p)$ equals 1 for the same reason. Plugging this into our calculation above, we see:

$$e(g_p^a g_q^a, g_p^b g_q^b) = e(g_p, g_p)^{ab} e(g_q, g_q)^{ab}.$$

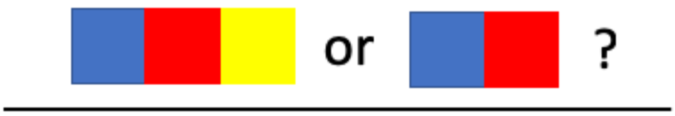
This means that the p and q order subgroups are "orthogonal" under the bilinear map, and computations are still happening in parallel modulo p and modulo q in the target group, tied together but not intermixing. It can be helpful to visualize this with different colors representing the mutually orthogonal subgroups: a group element can be abstracted to a collection of colors corresponding to its non-trivial subgroup elements, and the pairing preserves only the colors that are present in both inputs. For example, we'll let blue represent the presence of g_p^a for some $a \neq 0 \bmod p$ and red represent the presence of g_q^b for some $b \neq 0 \bmod q$, while the absence of blue/red implies the g_p/g_q component is g_p^0/g_q^0 respectively. Then the pairing relationship behaves like:

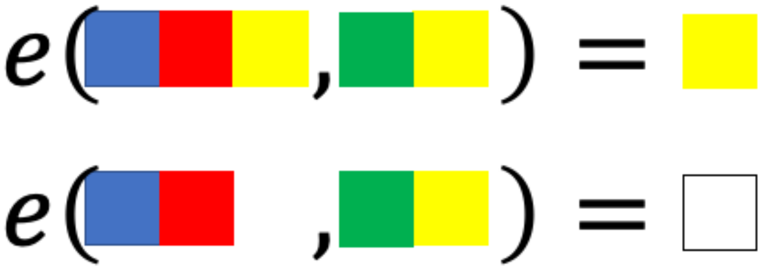


Hence pairing an element of G with an element like g_q in the bilinear map will yield 1 in the target group *if and only if* the q-part of the element of G is trivial. Hence, assuming we can efficiently recognize the identity element in G_T, giving out the element g_q (or any other element exclusively inside the order q subgroup) would break our formulation of the subgroup decision assumption. But note that giving the element g_p, or elements like $g_p^r g_q^t$, does not enable this attack.

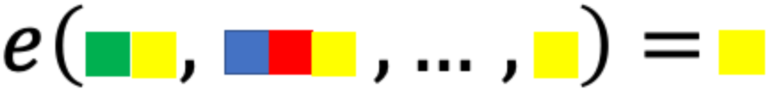
We can generalize this thinking to an arbitrary number of mutually orthogonal subgroups. Consider, for example, a bilinear group with an order N that is the product of 4 distinct primes: $N = p_1p_2p_3p_4$. We could define a plausibly hard subgroup decision problem as follows. We could take two subgroup combinations that we want to make it a challenge distinguish, e.g. the subgroup of order $p_1p_2p_4$ and the subgroup of order p_1p_2, and then give out generators and random elements from subgroup combinations that do not allow a trivial distinguishing attack on the challenge as a consequence of orthogonality under the bilinear map. In our example, we can give out a random element from a subgroup of any order *except* an order that is divisible by p_4 but not divisible by either p_1 or p_2. Note here that an element of order p_3p_4, for example, could be paired with the challenge element under the bilinear map and would yield the identity in the case that the challenge element was of order p_1p_2, but not if it was of order $p_1p_2p_4$.

If we visualize this now with four colors, blue for p_1, red for p_2, green for p_3, and yellow for p_4, we see how an element of order p_3p_4 (green and yellow) would distinguish our subgroup challenge:





We can also generalize this thinking to a multilinear group with various subgroups that are mutually orthogonal under the multilinear map. Let's imagine that our multilinear map takes k group elements and essentially computes one k-way multiplication in the exponent, landing in a new target group where further exponent multiplications are not feasible. A particular subgroup will contribute something non-trivial to the final target group element if and only if *all* of the k input elements have a non-trivial component in this subgroup. In other words, a product of k terms in a finite field is non-zero if and only if *all* of the k terms are non-zero.



If the only feature of the final target group element that we are interested in is whether it is fully trivial or not, then we can describe this functionality as an "OR of ANDs." Breaking this down, we see the final element in the target group is non-trivial if it has a non-trivial component in the first subgroup of the target group, *or* a non-trivial component in the second subgroup of the target group, *or* a non-trivial component in the third subgroup of the target group, etc. Furthermore, it has a non-trivial component in a given subgroup if the first input has a non-trivial component in this subgroup, *and* the second input has a non-trivial component in this subgroup, *and* the third input has a non-trivial component in this subgroup, etc.

The "OR of ANDs" functionality becomes quite powerful if we can support an arbitrary polynomial number of ORs and an arbitrary polynomial of ANDs. As an example, let's consider a threshold function f_t from the set of positive integers $< 2^n$ to $\{0, 1\}$, parameterized by a threshold $0 \leq t < 2^n$: $f_t(x) = 1$ if $x > t$ and $f_t(x) = 0$ otherwise. To express this functionality as an "OR of ANDs," we can consider the binary representations of x and t, which we can write as:

$$x = x_0 2^0 + x_1 2^1 + x_2 2^2 + \cdots + x_{n-1} 2^{n-1}$$
$$t = t_0 2^0 + t_1 2^1 + t_2 2^2 + \cdots + t_{n-1} 2^{n-1}.$$

Now $x > t$ if:

$$x_{n-1} = 1 \text{ AND } t_{n-1} = 0$$

$$\text{OR } (x_{n-1} = t_{n-1} \text{ AND } x_{n-2} = 1 \text{ AND } t_{n-2} = 0)$$

$$\text{OR } (x_{n-1} = t_{n-1} \text{ AND } x_{n-2} = t_{n-2} \text{ AND } x_{n-3} = 1 \text{ AND } t_{n-3} = 0)$$

$$\text{OR } \ldots \text{ AND you get the idea.}$$

What remains is to translate these kind of equality conditions into the presence of a particular subgroup in the final result of a multilinear map computation. In the context of obfuscation, it is convenient to think of t as a hard-coded constant in the definition of the function f_t, and we want to give out an "obfuscated" version of f_t that allows someone to compute $f_t(x)$ for any desired x. This on its own may not seem very useful, as one can use strategically chosen inputs for x to binary search for the hard-coded value of t, seemingly defeating the purpose of obfuscation. However, when this function f_t becomes merely a component of a more complicated function (as occurs in the arguments given in [GLW14, GLSW15]), things begin to get more interesting.

To "obfuscate" a function such as f_t, which is expressed as a sequence of OR's of ANDs of equality conditions on individual bits of the input x, we will give out $2n$ group elements, two for each bit of the input. We will use an n-linear group, with dedicated subgroups for each OR condition. For example, if we have a function like:

$$(x_2 = 1) \text{ OR } (x_2 = 0 \text{ AND } x_1 = 1) \text{ OR } (x_0 = 1),$$

we can use three subgroups. Let's denote their orders by p_1, p_2, and p_3 respectively. We'll let g_{p_1}, g_{p_2}, and g_{p_3} denote their respective generators.

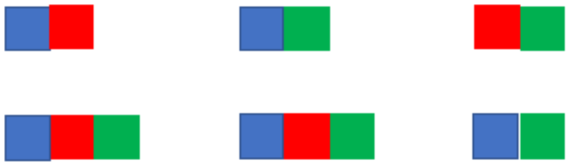
In this case, our input is only three bits long, so will give out six group elements: $E_{0,0}$, $E_{1,0}$, $E_{0,1}$, $E_{1,1}$, $E_{0,2}$, $E_{1,2}$. To evaluate our function on a input with bits x_0, x_1, x_2, an honest evaluator will compute the 3-linear map with inputs $E_{x_0,0}$, $E_{x_1,1}$, and $E_{x_2,2}$: if the result is the identity element of the target group, the output will be 0. Otherwise, the output is 1.

The first subgroup, generated by g_{p_1}, will contribute a non-trivial result to the final product if and only if $x_2 = 1$. To arrange this, we'll sample random components from $< g_{p_1} >$ for all of the elements $E_{0,0}, E_{1,0}, E_{0,1}, E_{1,1}$ and as well as for $E_{1,2}$, but we will use $g_{p_1}^0$ as the $< g_{p_1} >$ component of $E_{0,2}$. This way, if $x_2 \neq 1$, then the contribution of the p_1 subgroup to the final result will be trivial. Otherwise, it will be a random (non-zero whp) element of the p_1 subgroup of the target group.

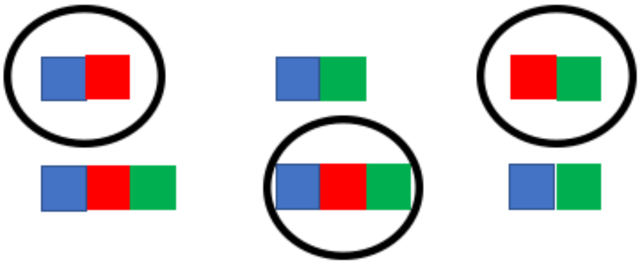
Similarly, the second subgroup, generated by g_{p_2}, will contribute a non-trivial result to the final product if and only if $x_2 = 0$ AND $x_1 = 1$. To arrange this, we'll sample random components from $< g_{p_2} >$ for the elements $E_{0,0}$ and $E_{1,0}$, as well as for $E_{0,2}$ and $E_{1,1}$. But we will use $g_{p_2}^0$ as the $< g_{p_2} >$ component of $E_{1,2}$ and $E_{0,1}$. This way, if $x_2 \neq 0$ or $x_1 \neq 1$, the contribution of the p_2 subgroup to the final result will be trivial. Otherwise, it will be a random (non-zero whp) element of the p_2 subgroup of the target group.

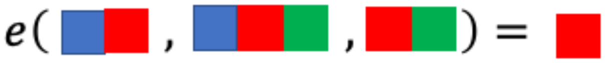
Finally, we'll sample random components from $< g_{p_3} >$ for all elements except $E_{0,0}$. Putting this all together, we'll have that the result of the multilinear map is the identity element in the target group if and only if the input bits x_0, x_1, x_2 fail to satisfy *all* of the three conditions that are strung together by ORs.

Visually, we can represent this as follows, with p_1 components represented by blue, p_2 components represented by red, and p_3 components represented by green:



To evaluate the function on the input 010, the computation would look like:





This shows how the satisfaction of the second clause translates into the non-triviality of the final result, but we note that if subgroup decision problems are hard in some multilinear group, this will not necessarily leak information to the evaluator about *which* clause was satisfied.

For this small example, all of this feels like overkill, since it is easy for someone with access to the obfuscated function to query it to learn the conditions that live inside of our ORs. But if we imagine a polynomial number of input variables and a polynomial number of polynomially many AND condtions (and say the word "polynomial" a polynomial number of times!), it seems that the obfuscation guarantee should become very meaningful.

But exactly what should the guarantee be? We've seen that virtual black box obfuscation is too much to hope for in general. But in the same paper where Barak et al. proved the impossibility of virtual black box obfuscation for all functions, they also suggested a notion of "indistinguishability obfuscation" that was not ruled out by their arguments.

Indistinguishability obfuscation is rooted in a long tradition of game-based security definitions in cryptography. The foundational definition of IND-CPA security for public key encryption, for example, requires that an attacker playing a game against an encryption challenger cannot tell the difference between an encryption of one message M_0 and an encryption of another message M_1, even when the attacker can decide on the values of M_0 and M_1 herself. This is what it means for two probability distributions to be "computationally indistinguishable": if an attacker whose computational resources are polynomially bounded is playing a guessing game between the two, she cannot do significantly better than flipping a coin. In the context of encryption, we have to place a few constraints in the rules of the game to make this work.

Namely, the attacker cannot know the randomness used during the probabilistic encryption algorithm, and also that the two messages M_0 and M_1 must be the same length (otherwise, the mere length of a ciphertext might give it away).

In obfuscation, we can imagine defining a similar guessing game, and we will see what kind of rules we need to put in place to make sure our attacker is sufficiently boxed in. We'll first let the attacker choose two functions, f_0 and f_1. The challenger will randomly set a bit $b \in \{0, 1\}$, and will return an obfuscation of f_b. The attacker can then examine this obfuscation of f_b for a polynomial amount of time, using polynomially-sized computational resources, and must then declare her best guess for whether b is 0 or 1. If she guesses correctly with probability no more than negligibly greater than $\frac{1}{2}$, we say that our obfuscation algorithm achieves *indistinguishability obfuscation.*

The first rule we must put in place is the scope and format of our function class: what universe will the attacker have to choose f_0 and f_1 from, and what format will be prescribed for describing her choice? For now, let's stick with the ORs of ANDs structure we were playing with above. Now a single set of AND conditions could be described in multiple ways if we allow redundancy, e.g $(x_0 = 1) AND (x_2 = 0)$ vs. $(x_0 = 1) AND (x_2 = 0) AND (x_0 = 1)$, so we'll stipulate that each input bit can be referenced at most once in each sequence of ANDs. Other than that, we will allow any combination of ANDs of equality conditions among the n input bits, and will allow an arbitrary polynomial number m of such combinations to be tied together by ORs. In the same spirit as requiring the messages M_0, M_1 to be the same length in the IND-CPA security game above, we will require that attacker's two functions f_0 and f_1 share the same values of n and m (so they have the same number of input bits, and the same number of ORs).

There is one remaining difficulty. What if the attacker chooses f_0 and f_1 in such a way that she knows of an input x where $f_0(x) \neq f_1(x)$? In this case, the requirement that an obfuscated function still function as a function[3] means that she will be able to run the obfuscated function on her helpful input x and see whether the output is consistent with f_0 or f_1 - thereby winning the guessing game every time!

It's not very clear how best to rule this out if we allow such inputs x to exist. So we will do something that may feel like cheating - we will insist that f_0 and f_1 must have the same output over *all* possible inputs. Only when this criterion is satisfied will we promise that the attacker will fail to reliably distinguish between obfuscations of f_0 and f_1.

But how will we know when this true? If an attacker describes f_0 and f_1 in terms of m ORs of many ANDs over n input bits, how will we know if $f_0(x) = f_1(x) \forall x$? The disappointing answer is: we won't unless we check exhaustively. This is why uses of indistinguishability obfuscation on route to proving other cryptographic properties typically resort to *complexity leveraging*: in order to extract meaning from the obfuscation guarantee, we will need to ensure that $f_0(x) = f_1(x) \forall x$ by checking this equality for all 2^n inputs x, and limit ourselves to values of n where we can afford to absorb this $\mathcal{O}(2^n)$ into our argument.

But what if we aim for a less ambitious guarantee? We can instead insist on pairs of functions f_0 and f_1 for which $f_0(x) = f_1(x)\ \forall x$ can be easily verified. If we stick to our disjunctive normal forms, it is not too hard to come up with examples of f_0, f_1 candidates where this is possible. Suppose that f_0 and f_1 are nearly identical, but differ by the addition of a single AND condition in a single clause:

$$\begin{aligned} f_0 &:= (x_1 = 0 \text{ AND } x_3 = 1) \text{ OR } (x_1 = 0 \text{ AND } x_2 = 0) \qquad (1) \\ f_1 &:= (x_1 = 0 \text{ AND } x_3 = 1) \text{ OR } (x_1 = 0 \text{ AND } x_2 = 0 \text{ AND } x_3 = 0) \end{aligned}$$

[3] pun intended!

Now, the second clause of f_1 has its support contained in the support of the second clause of f_0, but it is missing the inputs where $x_1 = 0$ AND $x_2 = 0$ AND $x_3 = 1$. However, we can see that this missing support is contained in the support of the first clause. Hence the set of all inputs where $f_1(x) = 1$ is identical to the set of all inputs where $f_0(x) = 1$.

If we abstract this example, we can formulate a relatively easy to check and sufficient condition to ensure that $f_0(x) = f_1(x)\ \forall x$. It is "safe" for us to add a condition $x_i = b$ to clause j of f_0 to form f_1 if there is another clause k satisfying the following two conditions

- For every $i' \neq i$, any condition on $x_{i'}$ in clause k appears in clause j as well.
- If there is a condition on x_i in clause k, it must be $x_i = 1 - b$.

In our example above, these conditions were satisfied with $i = 3$, $b = 0$, $j = 2$, and $k = 1$. More generally, these two conditions ensure that the additional support of the less restrictive j^{th} clause in f_0 is contained in the support of the shared k^{th} clause, ensuring that $f_0(x) = f_1(x)\ \forall x$.

This is stated in a slightly different notational framework in [GLW14], but indistinguishability obfuscation for pairs of DNFs f_0 and f_1 that are related in this way is what is deemed "intercolumn security." The "column" part of that name comes from a notation that expressed each conjunction as a "column" of a 3-dimensional matrix, and it was called "intercolumn" because the relationship *between* columns/conjunctions j and k is crucial here. The 3-dimensional matrix format is not particularly helpful for our purposes, so we will not detour to describe it.

Now, to use our subgroup decision framework in a multilinear group to try to achieve intercolumn security, we can let the number of clauses be the number of subgroups, and the degree of multilinearity can be set to the maximum number of equality conditions within a single clause.

To make things concrete and to put off the issue of implementing higher degrees of multilinearity in such a way that subgroup decision problems remain hard (a fundamental open research problem at the time of this writing), let's see what we can accomplish with just bilinear maps. The most straightforward application of intercolumn security in this would only involve two clauses (j and k) with at most two conditions each. But we can arrange things so that we can consider two clauses with an arbitrary polynomial number of equality conditions in each. This is because the negation of a DNF of this form becomes a new DNF with polynomially many clauses each containing two conditions. For example:

$$\neg\left((x_1 = 0 \text{ AND } x_3 = 1) \text{ OR } (x_1 = 0 \text{ AND } x_2 = 0 \text{ AND } x_3 = 0)\right)$$

is equivalent to:

$$\begin{aligned} &(x_1 = 1 \text{ AND } x_1 = 1) \\ \text{OR } &(x_1 = 1 \text{ AND } x_2 = 1) \\ \text{OR } &(x_1 = 1 \text{ AND } x_3 = 1) \\ \text{OR } &(x_3 = 0 \text{ AND } x_1 = 1) \\ \text{OR } &(x_3 = 0 \text{ AND } x_2 = 1) \\ \text{OR } &(x_3 = 0 \text{ AND } x_3 = 1) \end{aligned}$$

We chose to write that last one out to help illustrate the pattern, though since it is unsatisfiable, we could remove it. Intuitively, each clause represents one choice for how to fail to satisfy each of the original two clauses. If the two conditions involve the same variable, they either collapse to one condition (like the first line above, which reduces to simply $x_1 = 1$), or can be removed as unsatisfiable (like the last line above).

So how might we obfuscate a DNF like this with m short clauses? Let's try using a bilinear group G with m subgroups. We'll let $p_1, \ldots, p_m$ denote the orders of these subgroups. We'll let n denote our number of input bits, and for each input bit x_i we will construct two elements of G, one for use when $x_i = 0$ and one for use when $x_i = 1$. The distribution of these elements will be as follows:

- if $x_i = 1 - b$ *does not* appear as a condition in clause j, then the component in the subgroup of order p_j will be uniformly random for the group element representing $x_i = b$.
- if $x_i = 1-b$ *does* appear as a condition in clause j, then the component from the subgroup of order p_j will be trivial for the group element representing $x_i = b$.

When given these $2n$ group elements, a person can evaluate the DNF for a desired setting of the input bits by choosing the appropriate subset of n group elements representing that input, and then apply the bilinear map to each pair. The resulting $\binom{n}{2}$ elements of the target group can then be multiplied together, and the final result will be the identity element if and only if *all* of the clauses failed to be satisfied.

Correctness for this setup follows from the observation that the final element of the target group will be trivial if and only if each subgroup contribution is trivial, and the subgroup of order p_j will contribute something non-trivial if and only if and the j^{th} clause is satisfied. So far so good.

But security-wise, the careful reader should be concerned. But wait, you protest. A person doesn't *have to wait* to multiply all of the target group elements together before testing for the identity element. Assuming we have an efficient identity test procedure in the target group, it should work just as well on the individual pair outputs. This means we can learn something more than just if the overall DNF is satisfied - we can learn which pairs of elements contribute to its satisfaction. Furthermore, there is nothing that forces us to stick consistently to one value for each input bit. We could sometimes use the group element corresponding to $x_1 = 0$ for instance, and other times use the group element corresponding to $x_1 = 1$.

These kind of issues would surely rule out an all-encompassing security notion like VBB security, but it is not immediately obvious if they violate the much less ambitious goal of intercolumn security. In fact, let's see if we can sketch a proof of intercolumn security from instances of the subgroup decision problem.

We return to our running example of intercolumn security, labeled (1) above. In transitioning from f_0 to f_1, we added the condition AND $x_3 = 0$ to the second clause. If we follow the effect of this through our negation and transformation into a DNF with clauses containing at most two variables, we see that this resulted in the addition of two clauses, $x_1 = 1$ AND $x_3 = 1$ as well as $x_3 = 0$ AND $x_3 = 1$.

Let's first consider the addition of $x_1 = 1$ AND $x_3 = 1$. This will require us to add a new nontrivial subgroup component to the elements representing $x_1 = 1$ and $x_3 = 1$. Since this is the third clause in our list, we'll call this the subgroup of order p_3. We observe that there is *already* a clause that is satisfied when $x_1 = 1$ and $x_3 = 1$: namely the first clause, $x_1 = 1$ AND $x_1 = 1$. This is assured to us by the intercolumn security condition that for every $i' \neq i$, and condition on $x_{i'}$ in clause k appears in clause j as well. This means the subgroup corresponding to this clause will already have nontrivial components on the group elements representing $x_1 = 1$ and $x_3 = 1$! In our example, this is the subgroup of order p_1. So if we are given a generator of this subgroup, denoted g_{p_1}, and a challenge element T that definitely has a random component of order p_1 and may or may not have a random component of order p_3, we can use this challenge element T in creating the group elements representing $x_1 = 1$ and $x_3 = 1$, and effectively add the new clause in the case that T is of order $p_1 p_3$.

Now the one time we cannot apply this logic is when we are trying to add a clause like $x_3 = 0$ AND $x_3 = 1$, where *both* equality conditions refer to the same variable. But in this case, we are guaranteed by the second requirement of intercolumn security, that any condition on x_i in clause k must be $x_i = 1 - b$, that the conditions will be conflicting, and hence the clause is unsatisfiable, and can be omitted from our DNF representation.

Though we have only gone through an example here, it is fairly straightforward to extend this argument and make it rigorous, hence proving that intercolumn security follows from subgroup decision assumptions in a bilinear group, for formulas expressed as a single OR of strings of AND conditions, each containing polynomially many equality constraints on individual input bits.

Taking our reasoning a bit further, we might begin to suspect we don't even need a bilinear group. Afterall, an honest evaluator of the obfuscated boolean formula will only ever pair elements from the given set of $2n$ elements, and there are only $2n(2n-1)/2 = \mathcal{O}(n^2)$ pairs of group elements to throw into the bilinear map. Since this is a polynomial anyway, we could actually preprocess all of these pairings, and give the evaluator the set of resulting target group elements instead. It seems clear that this change should not introduce insecurities, as we are only giving the evaluator a polynomial number of things that she could easily compute herself from what we were previously giving her.

This is the kind of sanity check that can ironically make us a little nuts. What is really going on here? The subgroup decision problems in the target group become pretty basic - essentially we are just requiring that a random element from a subgroup in G_T (aka the target group) be indistinguishable from a random element of the whole group G_T, and there is no pairing to complicate things. In this case, there is no longer any need to stop specifically at two strings of AND conditions glued by a single OR. Why not three strings of AND conditions glued together by two ORs ? If we preprocess all the triples of input bits, we can hope to provide $\mathcal{O}(n^3)$ elements of a regular group that will be sufficient for evaluation.

Clearly this approach does have a limit - we won't get beyond polylog(n) ORs this way, as the preprocessing and output size will become superpolynomial beyond that point. But still, we might imagine that a constant or polylogarithmic number of ORs will still cover some non-trivial functionalities. So - yay? We've achieved something substantial?

You might pick up on the *subtle* hint of foreboding in those question marks. This is exactly the kind of moment every graduate student dreads. That moment where the pieces of your proof all seem to finally fall into place, and everything seems to work, and then you say to yourself: "ok, so let's sanity check that this is all as a meaningful as I think it is." When that happens in real life, I recommend you stop and sleep on it. Enjoy your moment of believing it all works and is as impressive as you imagine it could be! Revel in that moment. *Linger* in it. Loiter as long as you can, until that rare moment of intellectual satisfaction forcibly expels you from its glow.

In some small percentage of such times, the dreaded expulsion will never come. The axe will never fall, and you will remain in the garden of Eden, your beautiful theorem secure in its fundamental glory.

This is not one of those times.

Chapter 5: The Maddening Sanity Check

It is a general feature of life that there is rarely only one way of doing anything. In cryptography, this seems surprisingly less true than it feels like it should be. Cryptographers have, after all, dedicated many decades of effort to coming up with candidates for "one-way" functions, where computing them in the intended direction is relatively easy, but inverting them is unreasonably hard. And yet, we only have three basic categories that have risen to broadly accepted prominence:

1. multiplication of large primes (easy) vs. factoring into large primes (hard)
2. computing a group operation (easy) vs. the discrete logarithm problem (hard for some groups)
3. matrix multiplication (easy) vs. finding "short" solutions to linear equations (hard)

The first candidate for a one-way function is likely familiar to you as the backbone of the RSA public key encryption and signature schemes. The second candidate (which is a actually a category of candidates, depending on your selection of group) is what we've focused on in the last few chapters. (Note that problems like subgroup decision problems in a group can only be hard if the discrete logarithm problem is hard, so this being a one-way function is a necessary though not known to be sufficient condition for everything we've been building in groups so far.) But the third one-way function category deserves a bit of exposition at this point.

Linear algebra over $\mathbb{Z}_p$ is similar to linear algebra over the rational numbers, or the real numbers, or the complex numbers, in many ways. Matrix multiplication is efficient, and matrix inversion is efficient as well. But the analogies to other fields start to feel a bit less natural when we impose a notion of "smallness" on $\mathbb{Z}_p$. Intuitively, we have a gut sense of what we mean by a *small* real number, at least comparatively. A real number like 2.3334 is smaller than 10.3029333..., and we can extend our intuitive notion of smallness to vectors over the real numbers by choosing any one of many natural-feeling or not-so-natural-feeling norms to measure the "length" of a real vector, such as the Euclidean norm.

If we think about this too long though, it can start to feel arbitrary. What's so special about the point 0 on the real number line? It's just a point like any other point in some ways. Why should numbers near it be labeled "small" (in absolute value), and numbers far away from it be labeled "large?" We can declare the equivalence class of 0 modulo p to be similarly special, and represent the elements of $\mathbb{Z}_p$ as $-\frac{(p-1)}{2}, \dots, -1, 0, 1, \dots, \frac{p-1}{2}$ so that some neighborhood from $-X$ to $+X$ that is centered around 0 can be declared to contain the "small" elements of $\mathbb{Z}_p$. In some sense, we can visualize the 0 in $\mathbb{Z}_p$ to play the same role as the 0 on the real number line, except that we have now wrapped the line around a circle, bounding all distances by $\frac{p-1}{2}$.

It is this notion of "small" that doesn't play nicely with matrix inversion (or more basically, with division), and hence provides an opportunity for a one-way function. If we take a vector $x \in \mathbb{Z}_p^m$ with "small" entries, and a matrix A chosen uniformly at random in $\mathbb{Z}_p^{n \times m}$, then $f_A(x) := Ax$ defines a candidate family of one-way functions from a domain of vectors in $\mathbb{Z}_p^m$ with small entries to the range of all vectors in $\mathbb{Z}_p^n$. Why is inverting this potentially hard for some reasonable ranges of parameters? Well, first we'll choose m to be larger than n (typically $m > n \log p$), so that A is not invertible, and furthermore, treating Ax as a bunch of linear equations in the unknown entries of x is an underdetermined system. Now, it isn't hard to find *some* solutions to this underdetermined system of linear equations. But finding a very "small" solution is believed to be hard.

A related computational problem is the "Learning with Errors Problem," where the task is to solve an *over*determined system of equations that has been perturbed by small noise values. To set up the problem, we choose a secret vector $s \in \mathbb{Z}_p^n$, a uniformly random matrix $A \in \mathbb{Z}_p^{n \times m}$, and a "small" noise vector $e \in \mathbb{Z}_p^m$, whose entries are each selected independently from a distribution highly concentrated near 0 on $\mathbb{Z}_p$. The problem statement is then to distinguish between two distributions over $\mathbb{Z}_p^{n \times m} \times \mathbb{Z}_p^m$: the distribution of A, $A^t s + e$ and the distribution of A, r where r is a uniformly random vector from $\mathbb{Z}_p^m$. Naturally the problem varies a bit as you consider various distributions for s and for e. For simplicity, we'll think of s as being uniformly random over $\mathbb{Z}_p^n$ (though it has been shown that s can be taken to be "small" itself, but we won't need this).

Let's first observe that without the noise vector e, the problem would be easy. $A^t s$ is a system of m linear equations in $n < m$ unknowns, so it will exhibit ample efficiently recognizable structure (since A itself is not kept secret). But with e added in the mix, it's not at all clear what to do. Any known linear dependency between the rows of A^t will likely involve some large coefficients, and multiplying those by even the small elements of e will still produce potentially large, random elements that will obscure the structure we are trying to detect. That's the pesky thing about small noise - it's small when you add it, but if you multiply it by something big, it runs amok. Mixing noise with multiplication or division is like giving a toddler some sugary candy - suddenly the small thing exudes an uncontrollable scale of energy.

We could try to guess e by brute force, but even a small number of possibilities for each entry yields exponentially many possible e vectors as a function of m. If we were to take entries of e strictly from the set of two possible values $\{0, 1\}$ modulo p, we could observe that each entry of e is a zero of the polynomial $x^2 - x$, and hence use a linear equation like

$$a_{1,1}s_1 + a_{2,1}s_2 + \cdots + a_{n,1}s_n + e_1 = c_1,$$

where the a's and c's are known constants and the s's and e's are unknowns, to derive a quadratic equation like:

$$(a_{1,1}s_1 + a_{2,1}s_2 + \cdots + a_{n,1}s_n - c_1)^2 - (a_{1,1}s_1 + a_{2,1}s_2 + \cdots + a_{n,1}s_n - c_1) = 0.$$

We can then re-conceptualize quadratic terms like $s_i s_j$ to be our new variables, so we now have a linear equation in $\mathcal{O}(n^2)$ unknowns. If m is on the order of n^2, we might be able to solve such a system. Since m is on the order of $n \log p$, we won't have to increase the number of possibilities for each e_i by much until we force a number of new linear unknowns that far exceeds the number of equations we have, rendering this linearization approach ineffective.

In some sense, adding the random noise e to the structured vector $A^t s$ hides the structure without destroying it, much like putting the structure in the exponent of a group where the discrete log problem is hard. Let's consider how we might build a subgroup-like structure in this setting to get a new implementation of our "ORs of ANDs" obfuscation approach.

The seemingly most natural place to look for "subgroup" analogs inside a world driven by matrix operations is subspaces. We will replace our concept of a group element with nontrivial components in a several subgroups with the concept of a vector with nontrivial components in several subspaces, e.g.:

$$A^t s + B^t v + C^t w + e$$

is a vector with components from three subspaces (namely the ranges of A^t, B^t, and C^t), and the noise vector e makes it presumably hard to tell which subspaces are present, even if you know the matrices A, B, and C.

To map our "ORs of ANDs" boolean functions onto group behaviors, we previously used group elements for each input bit value and allowed multiplication in the exponent to implement the AND parts, while addition in the exponent and subgroup orthogonality implemented the ORs. Each subgroup was meant to encode one clause. Now we will try having subspaces encode clauses, and we will have an $m \times m$ matrix correspond to each input bit value.

There is one unfortunate collision of notation that we need to navigate here. In the world of Learning with Errors (abbreviated LWE), n typically denotes the smaller dimension of $n \times m$ matrices. In boolean functions as we have been using above, n typically denotes the number of bits in the input to a function. Since we will only need a fairly small example to illustrate our point here, we will keep with the LWE convention and have n continue to refer to a matrix dimension, and we won't give an abstract name to the number of input bits at all. We'll make it three. And we'll have just two clauses:

$$(x_1 = 1 \text{ AND } x_2 = 1)$$
$$\text{OR } (x_2 = 0 \text{ AND } x_3 = 0).$$

We'll pick two subspaces, spanned by $A, B \in \mathbb{Z}_p^{n\times m}$, to correspond to these two clauses, and we'll have 6 matrices, denoted $M_{0,1}$, $M_{1,1}$, ..., $M_{0,3}$, $M_{1,3}$, corresponding to the possibilities $x_1 = 0$, $x_1 = 1$, ..., $x_3 = 0$, and $x_3 = 1$ respectively. To enable evaluation of the intended boolean function, we will give an evaluator these six matrices as well as two "bookend" vectors, v_{start} and v_{end}, and evaluation will proceed by taking v_{start}, multiplying by the appropriate $M_{b_{i},i}$ matrices corresponding to the desired input, and then finally multiplying for v_{end}. For example, to evaluate on the input $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$, one would compute:

$$v_{end} M_{0,3} M_{1,2} M_{0,1} v_{start}.$$

Since order matters for matrix multiplication, we'll use as a convention for now that the input bits are applied in order right to left as we wrote here. To make all of the dimensions work out, we will have v_{start} be an $m \times 1$ vector, and v_{end} be a $1 \times m$ vector. To turn our final output (which is a scalar in $\mathbb{Z}_p$) into a boolean result, we will have a threshold: if it is "small", we'll output 0. Otherwise, we'll output 1.

What remains is to decide how to choose distributions for all of the matrices and vectors that reflect our boolean function. For v_{start} we will keep it simple and make its distribution $A^t s + B^t w + e$, where s, w are uniformly random vectors in $\mathbb{Z}_p^n$, and e is a noise vector in $\mathbb{Z}_p^m$. So we will start with both subspaces being present. Intuitively, we will want to arrange things so that our $M_{b,i}$ matrices will zero out any surviving contributions from these original subspaces corresponding to clauses that are violated when $x_i = b$. For clauses that can be satisfied when $x_i = b$, $M_{b,i}$ should preserve their influence. We will also need $M_{b,i}$'s to interact with the noise in a controlled way so that the effect of the noise does not ultimately swamp the result of our computation. To accomplish this, we'll hope to choose them all to have small entries.

Let's think about what we might want to accomplish with the distribution of the matrix $M_{0,1}$ The value $x_1 = 0$, fails to satisfy the first clause, but does not fail the second. So we can choose $M_{0,1}$ such that $M_{0,1}A^t$ is an $m \times n$ matrix of all 0's in $\mathbb{Z}_p$. This will effectively kill the initial contribution of A^t, much like multiplying by 0 in the exponent of a particular subgroup. In contrast, we will not choose $M_{0,1}$ to kill the contribution of $B^t w$, so $M_{0,1}B^t$ should not be all 0's. Finally, we want to choose the entries of $M_{0,1}$ to be "small", so that $M_{0,1}e$ remains a noise term of controlled magnitude.

The value $x_1 = 1$ is consistent with both clauses, so we will not want $M_{1,1}$ to have either the range of A^t or the range of B^t in its kernel. But we might worry about the ranks of our matrices giving away unnecessary information about the specifications of our clauses, so it seems helpful to declare that all of the $M_{b,i}$ matrices will have a common rank (which should be $m - 2n$ or lower), in order to have "room" in their kernels for the relevant subspaces.

Let's think through what happens if we choose $M_{0,1}$ to be a random matrix of rank $m - 2n$, up to the constraint that it has the range of A^t contained in its kernel and small entries, and we choose $M_{1,1}$ to be a random matrix of rank $m - 2n$ with small entries (so with high probability, $M_{1,1}A^t$ and $M_{1,1}B^t$ will be non-zero, rank n matrices). Next we will need to decide on distributions for $M_{0,2}$ and $M_{1,2}$. At this stage, we have to account for the fact that these matrices won't be acting on A^t and B^t directly, but will be acting on $M_{0,1}A^t$ and $M_{0,1}B^t$, or on $M_{1,1}A^t$ and $M_{1,1}B^t$, depending on the value of the first bit if we are in the middle of evaluating some input.

Since $x_2 = 0$ is consistent with the second clause but not the first, we might choose $M_{0,2}$ so that the ranges of $M_{0,1}A^t$ and $M_{1,1}A^t$ are both contained in its kernel. In this particular example, $M_{0,1}A^t$ is just 0s, but that need not be the case in general. If we want to extend this approach in the natural way to arbitrary boolean clauses over many variables, we will quickly drown in the exponentially growing number of constraints that we will accumulate as we need to account for all the possible value combinations of all of the previous input bits.

So it seems clear that some new ideas would be needed to get an approach like this to work for arbitrary DNFs with polynomially many polynomially sized clauses, but perhaps if we limit either the number of clauses or the size of clauses as we did to get something working with bilinear groups, we could afford to blow up the dimensions of our space exponentially in the bounded parameter. But can the learning with errors problem really serve as a general substitute to the subgroup decision problems we were using before?

We might worry about the construction of the matrices $M_{b,i}$ with small entries. In the typical presentation of LWE, there are no short vectors to be found. We are given A and either $A^ts + e$ or a random vector r, and if we knew even one short vector z such that zA^t was all 0s, we could use it to distinguish $A^ts + e$ from a random r because $z(A^ts + e)$ would be small, while zr would not (with high probability).

But there is a variant of LWE called k-LWE [LPSS14] that looks a little more subgroup-y. Instead of one matrix A, we will have two matrices A and B, and the challenge will be to distinguish A^ts+e from A^ts+B^tv+e. What's interesting here is what we can also be given: in addition to being given A and B, we can also be given a few short vectors (k is the name for the number of them, hence the name k-LWE) that are orthogonal to *both* A^t and B^t. Let's work with $k = 1$ for simplicity. In this case, we are given one short vector z such that $zA^t = zB^t = 0$, where 0 here means a vector of all 0 entries over $\mathbb{Z}_p$. Now, this vector z doesn't break the challenge like it did before, since $z(A^ts + e)$ and $z(A^ts + B^tv + e)$ are both small.

If we get greedy and ask for enough vectors z to form a full basis of the space of vectors orthogonal to both A^t and B^t, however, things fall apart. The trouble starts with the observation that we can always efficiently find *some* vector v which satisfies $vA^t = 0$ but not $vB^t = 0$. If we have a full basis of short vectors z such that $zA^t = zB^t = 0$, then we can adjust v by adding and subtracting multiples of such vectors to make v short, while still preserving $vA^t = 0$ and $vB^t \neq 0$. Intuitively, having a full short basis of such z's gives us good precision control to make v short in all dimensions. If we only have a more limited collection of short z's, we will be able to make v short with respect to some entries/dimensions, but others will stubbornly remain large (at least using known methods bounded by polynomial time).

Putting this all together, the k-LWE problem really does look like a subgroup decision problem if you squint just right. The additional short vectors that don't form a basis are like additional group elements from various combinations of subgroups that yield the identity element in the target group when paired with *either* distribution of the challenge element.

But there is something a bit weird about the framework we've set up to build with k-LWE. We plan to use the challenge term to make v_{start}, and the short z vectors to help us make the $M_{i,b}$ matrices, but we will never give out the underlying matrices A and B as part of our obfuscation. Why is this weird? Well, if we remove A and B from what is presented in the k-LWE problem, we are left with:

$$\text{Given short vectors } z_1, \ldots, z_k \text{ such that } z_iA^t = z_iB^t \ \ \forall i,$$

$$\text{distinguish } A^ts + e \text{ from } A^ts + B^tv + e.$$

The influence of A and B here is so weak, that really these two distributions become the same. In either case, all we have is k short vectors $z_1, \ldots, z_k$, and a "challenge" vector whose dot product with each z_i is small. We can define A and B after the fact to explain the structure either in one way or the other, but this doesn't change the distribution.

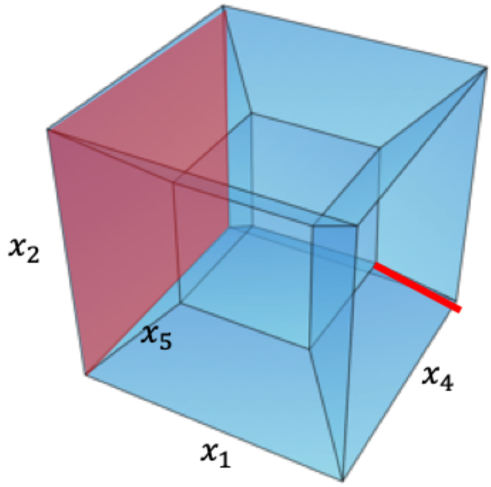
And this is where the depressing realization hits: intercolumn security for DNFs with a polylog bound on either the number of input bits or the length of clauses is not a goal that stands as a shining example of how discrete log cryptography and lattice cryptography can similarly be leveraged, but rather a goal that can be achieved *without either*. So yes, our approach likely works! And it is wholly unnecessary. Actually we can get intercolumn security for these bounded boolean functions information-theoretically.

This turns out to be an example of a fairly general phenomenon: indistinguishability obfuscation for a family of function representations can be achieved information-theoretically whenever there is an efficiently computable canonical form. If you can take any two different representations of the same function within some specified family of representations, and map each to a canonical form that is unique per function, then indistinguishability obfuscation trivially follows. The output of the obfuscation will simply be the canonical form, and the information of which of the equivalent (non-canonical) representations you may have started with will be entirely lost.

So how might we find an efficiently computable canonical form for the functions and representations we have been working with? Let's reconsider the basic form of our functions as DNFs with a single OR of polynomially many conditions strung together by ANDs, e.g.

$$(x_1 = 0 \text{ AND } x_3 = 1 \text{ AND } x_5 = 0) \text{ OR } (x_1 = 1 \text{ AND } x_2 = 0 \text{ AND } x_3 = 1 \text{ AND } x_4 = 1).$$

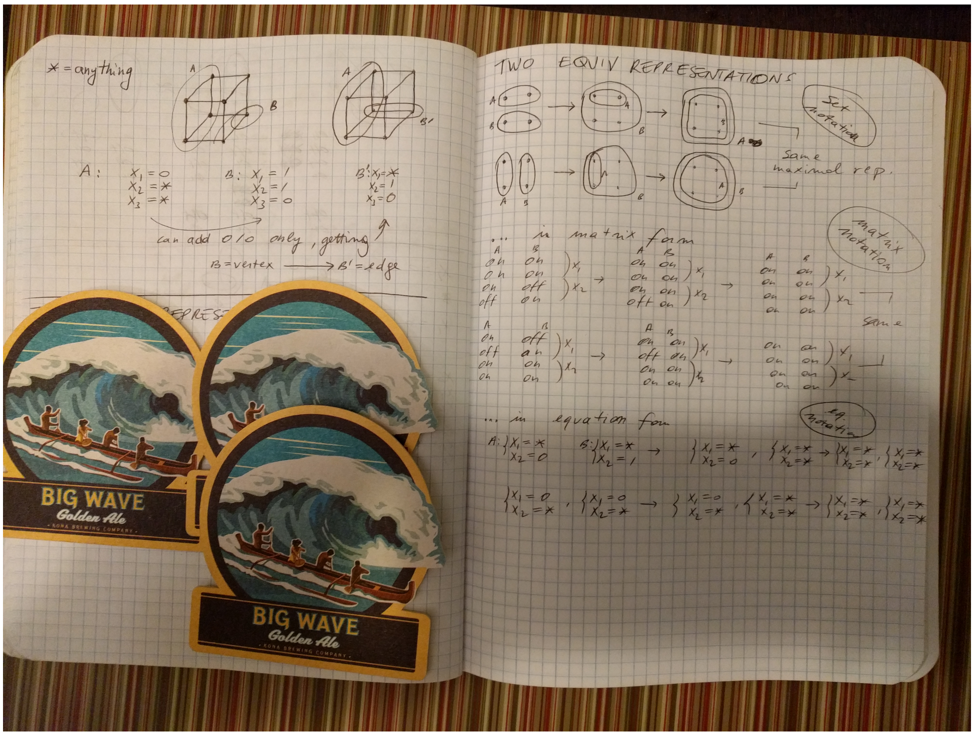
Let's think about the *support* of one of these clauses, i.e. the inputs that satisfy the clause. In the 5-dimensional hypercube corresponding to the 2^5 possible input value combinations for $x_1, \ldots, x_5$, the set of points which satisfy a single clause of equality conditions strung together by ANDs is itself a smaller dimensional hypercube inside the larger one. For example, the inputs where $x_1 = 0$, $x_3 = 1$, and $x_5 = 0$ form a 2-dimensional square defined by the degrees of freedom, x_2 and x_4, whose values are unconstrained. Similarly, the support of the second clause above is a 1-dimensional hypercube, as x_5 is the only unconstrained input. The union of these two lower dimensional hypercubes forms the support of the DNF with these clauses. In this case, the two sub-cubes don't intersect, and the number of points in the union is simply the sum of the number of points in each of these. To make this easier to see, we will visualize this inside the 4-dimensional hypercube, removing the constant variable $x_3 = 1$:



Some people like to think about objects like these geometrically. Such people can visualize sub-faces of sub-cubes fusing together to form higher dimensional sub-cubes, and they might, after a magical moment of meditation with their eyes closed suddenly declare: “Oh yes, I see it now! These unions of two sub-cubes have a canonical form! Given any such collection of points, I can give you a canonical description of it as an OR of two AND clauses, and hence indistinguishability obfuscation becomes trivial!” Such people are called “geometers.”

But when some other people close their eyes to try to visualize such geometric wonders, they see... nothing. Such people are called “algebraists.”

One of our fellow travelers on this journey, Valerio Pastro, is a geometer, and at this point he became convinced that a canonical form could be designated by finding a sub-cube of maximal dimension. He took this picture of his notes at the time:



Amusingly, he covered up some “distasteful” messiness in his thinking with some Big Wave Golden Ale coasters. This is typical of geometers. They are drawn to clean lines and smooth surfaces, and they seek to push under the rug the sweaty and gritty byproducts of mental struggle.

Algebraists, however, sometimes revel in devilish sequences of messy reductions, laundry lists of tackily specific case analyses, and otherwise rub their greasy paws all over a geometer’s precious canvas. And this is precisely what we will now do to Valerio’s elegant intuition. We will tear it apart in order to understand it - a bit of an autopsy for a proof.

Let’s return to our guiding example:

$$(x_1 = 0 \text{ AND } x_3 = 1 \text{ AND } x_5 = 0) \text{ OR } (x_1 = 1 \text{ AND } x_2 = 0 \text{ AND } x_3 = 1 \text{ AND } x_4 = 1).$$

We can first observe that both clauses require $x_3 = 1$, so really all of the action here is taking place within a 4-dimensional hypercube, and we can essentially ignore this condition (and just tack it on to all clauses once we arrive at a canonical form expressing the conditions on the other variables. In a similar fashion, we can ignore any variables that never appear in any of our equality constraints. In our example, this just means we don't need to be concerned if there is an unmentioned x_6, etc.

So let's rewrite the reminder of our functionality without the common $x_3 = 1$ condition:

$$(x_1 = 0 \text{ AND } x_5 = 0) \text{ OR } (x_1 = 1 \text{ AND } x_2 = 0 \text{ AND } x_4 = 1). \tag{2}$$

Now, there are two situations that are possible for each variable. Some variables, like x_2, x_4, and x_5 above, appear only once, invoked in an equality condition in only one of the two clauses. While one variable, x_1 above, is invoked on both sides, with contrasting conditions.

The size of the support of each clause is inversely related to its number of constraints. In our case, this means that the clause with two conditions, $x_1 = 0$ AND $x_5 = 0$, has a larger support than the clause with three conditions, $x_1 = 1$ AND $x_2 = 0$ AND $x_4 = 1$.

We might ask, what's the fewest number of conditions a clause can have if its support is to be contained in the set of points that satisfy our expression over x_1, x_2, x_4, and x_5? Is it possible, for instance to have a clause with only *one* condition whose support is a subset of our support?

What might such a clause be? Well, it can't be $x_1 = 0$ or $x_1 = 1$. We can prove this by contradiction: if it were $x_1 = 0$, all of its support would have to be contained in the support of our first clause above, $x_1 = 0$ AND $x_5 = 0$. But we can't ensure such a containment without the condition $x_5 = 0$! Similarly, it can't be $x_1 = 1$, since then its support would have to be contained in the support of the second clause above, so we can't skimp on any of the further conditions there either.

So if we are going to have a clause with only one equality condition whose support is contained in the support of (2), it must involve one of the variables x_2, x_4, or x_5. Let's suppose it involves x_2. Well, if we try $x_2 = 1$, we are in conflict with the second clause of (2), and are then stuck needing all of the conditions of the first clause. So we might instead try $x_2 = 0$. To see that this also fails, we note that all of the points where $x_2 = 0$ and $x_1 = 1$ must then be covered by the support of the second clause, because they are in the support of $x_2 = 0$, but cannot be covered by the first clause. But they are not fully covered, as we are missing the points where $x_2 = 0$, $x_1 = 1$, and $x_4 = 0$.

Similar reasoning can be followed to conclude that a clause with a single condition involving x_4 or x_5 also will not have its support contained in the support of (2). And soon enough, if you spend enough hours scribbling boolean DNFs on napkins by candlelight, while your roommates, family members, and beloved labrador retrievers implore you to stop and go to bed, you will convince yourself that efficiently computable canonical forms for 2-clause DNFs exist, and hence all of that beautiful work we did to obtain intercolumn security for obfuscating them from subgroup assumptions or from LWE was utterly pointless.

We strongly suspect that going from 2 clauses to 3, or to 4, or to 42, or to a poly-logarithmic number does not qualitatively change things. But frankly we are too depressed to work through such an argument.

Chapter 6: The Subset Sum Problem Problem

Instead, like Sisyphus, we find ourselves once again at the beginning. Tossed around by the winds of fate, fooled by fog into walking in circles, we find that when the dust settles, we seem no closer to our goal of constructing provably secure obfuscation for complex functions. Knowing more of what we don't know may feel like little comfort.

Wasn't life so much simpler when we were only trying to obfuscate a point function? Remember our gentle, function friend,

$$P_\alpha(x) = \{1 \text{ if } x = \alpha,\ 0 \text{ otherwise?}$$

It feels like so long ago now, but we discussed a few different ways of accomplishing that simple goal. We considered using a hash function H, and publishing the value $H(\alpha)$ along with a description of H. We considered using a group $G = \langle g \rangle$ where the discrete log problem is hard, and publishing g^α along with a description of G and its (efficiently computable) group operation. And finally we considered using a matrix branching program, a method which immediately generalized to a very rich class of functions.

But if we reign in our wider ambitions and look more narrowly for a new way to obfuscate P_α, we see that the matrix operations in our branching program for P_α (arranging the matrix product to match I only when the input bits all match the bits of α) are overkill. We can actually do this with just linear operations, no multiplication required.

To see this in its most simple form, imagine $\alpha = 000...0 \in \mathbb{Z}_p^n$. In this case, we can sample random values $s_{b,i} \in \mathbb{Z}_p$ for each $b \in \{0,1\}$ and $i \in [n]$, except that $s_{0,n} := -s_{0,1} - s_{0,2} - \cdots - s_{0,n-1}$. Now the $2n$ values $s_{0,1}, s_{1,1}, \ldots, s_{0,n}, s_{1,n}$ will be almost entirely random, except for the *one* structured sum,

$$s_{0,1} + s_{0,2} + \cdots + s_{0,n} \equiv 0 \bmod p.$$

To evaluate the function on a input $x_1 \ldots x_n \in \{0,1\}^n$, we compute the sum $\sum_{i=1}^n s_{x_i,i}$ and output 1 is this is $\equiv 0 \bmod p$, and output 0 otherwise.

If this one piece of structure was embedded in an unknown location instead of at 00...0, it's plausible that it would be infeasible to find, at least for values of n where 2^n is an impractically large quantity of computational work. This is in fact very close to a computational assumption known as the "subset sum" assumption. The only difference here is that the $s_{0,i}, s_{1,i}$ are organized in pairs. A more typical statement of the subset sum assumption says that is hard to distinguish between N uniformly random values in $\mathbb{Z}_p$, and N values that are uniformly random up to the constraint that a hidden subset of them sum to 0.

More formally, let's define two distributions on N elements of $\mathbb{Z}_p$:

$$\mathbb{D}_1 := \text{ sample } z_1, \ldots, z_N \text{ uniformly at random from } \mathbb{Z}_p$$

$$\mathbb{D}_2 := \text{ sample } \vec{v} \in \{0,1\}^N \text{ uniformly at random, and let } S \subseteq [N] \text{ denote the set of indices } i$$
$$\text{such that } v_i = 1. \text{ Sample } z_1, \ldots, z_N \text{ uniformly at random from } \mathbb{Z}_p$$
$$\text{subject to the constraint that } \sum_{i \in S} z_i \equiv 0 \bmod p.$$

The subset sum assumption asserts that these two distributions are computationally indistinguishable. In order to massage this into a direct statement about our obfuscation of a point function using the pairs of values $s_{0,i}, s_{1,i}$, it may help to consider a slight variation of these distributions:

$$\mathbb{D}_3 := \text{ sample } s_{0,1}, s_{1,1}, \ldots, s_{0,n}, s_{1,n} \text{ uniformly at random from } \mathbb{Z}_p$$

$$\mathbb{D}_4 := \text{ sample } \vec{v} \text{ in } \{0,1\}^n \text{ uniformly at random. Sample } s_{0,1}, s_{1,1}, \ldots, s_{0,n}, s_{1,n}$$
$$\text{subject to the constraint that } \sum_{i=1}^n s_{v_i,i} \equiv 0 \bmod p.$$

If we assert that the distributions $\mathbb{D}_3$ and $\mathbb{D}_4$ are computationally indistinguishable, then it is an immediate consequence that our obfuscation of a point function *for a uniformly random point* is computationally indistinguishable from a constant function. This is a satisfying obfuscation guarantee, and it is tantalizingly close to the typical assertion that $\mathbb{D}_1$ and $\mathbb{D}_2$ are computationally indistinguishable.

There is just one small missing piece if we want to obtain our obfuscation guarantee from the more typical version of the subset sum assumption - an argument that computational indistinguishability for $\mathbb{D}_1, \mathbb{D}_2$ implies computational indistinguishability for $\mathbb{D}_3, \mathbb{D}_4$. But is this true? It's certainly not obvious. If we simply set $N = 2n$, we see that distributions $\mathbb{D}_1$ and $\mathbb{D}_3$ are identical, but distributions $\mathbb{D}_2$ and $\mathbb{D}_4$ are not: a random subset of $[N]$ as sampled in $\mathbb{D}_2$ is not likely to respect the pair structure of the subset of chosen in $\mathbb{D}_4$.

To get ourselves in the right mindset to understand this disconnect, we imagine we have a talented but highly particular friend who is very good at distinguishing $\mathbb{D}_3$ from $\mathbb{D}_4$. But if we give present her with any *other* task, she may become annoyed and behave arbitrarily. The task we want to get to perform for us is distinguishing $\mathbb{D}_1$ from $\mathbb{D}_2$, but we cannot ask her to do this directly. We must instead massage our desired task into the format of $\mathbb{D}_3$ versus $\mathbb{D}_4$, so that our highly particular friend does not get annoyed.

We suppose we are given a sample of $z_1, \ldots, z_N$, and we are supposed to guess whether our sample comes from distribution $\mathbb{D}_1$ or distribution $\mathbb{D}_2$. We can try to translate our problem into a form our talented friend will understand. We'll set $n = N$, and sample our own uniformly random values $r_1, \ldots, r_{n-1} \in \mathbb{Z}_p$, and set $r_n := -r_1 - r_2 - \cdots - r_{n-1}$. For each index i from 1 to n, we choose a random value $b_i \in \{0, 1\}$. We set:

$$s_{b_i,i} := r_i, \quad s_{1-b_i,i} := r_i + z_i \quad \forall i \leq N$$

This set up has an appealing feature: if there is some subset sum structure inside in the $z_i's$, it will now become a subset sum structure that respects the convention of choosing one element from each slot. (You can choose the $r_i + z_i$ values for the slots corresponding to the hidden subset sum in the z_i's, and the r_i values for the other slots. The r_i's will ultimately cancel out either way.)

But this does not give us distribution $\mathbb{D}_4$, because there is more correlation here than should be expected in $\mathbb{D}_4$. For example, suppose that $z_1 + z_2 + z_4 \equiv 0 \bmod p$. Then we will have:

$$r_1 + r_2 + r_4 \equiv (r_1 + z_1) + (r_2 + z_2) + (r_4 + z_4) \bmod p,$$

whereas in the distribution $\mathbb{D}_4$, these elements would be jointly uniformly distributed, and correlation would only appear when considering elements from *all* indices from 1 to n.

The challenge here is coming from the fact that we don't know which slots correspond to the hidden subset sum among the z_i's (when the z_i's come from distribution $\mathbb{D}_2$). So we have to make sure the extra randomization of the r_i's cancels out *regardless* of which slots correspond to the hidden subset sum. This seems very difficult (impossible?) to do without inducing unwanted correlations. And even if we overcome this challenge, all we get for it is yet another way to obfuscate a point function. Not very exciting.

Chapter 7: The Generic Conclusion

If we step back a moment, we realize there may be other ways to embed structure that can be agnostic to exactly which elements we choose from at least some of the slots, and this can get us beyond point functions. Let's consider instead the task of pattern matching with wildcards. More precisely, let's consider functions $f : \{0,1\}^n \rightarrow \{0,1\}$ whose supports are described by templates like: $001 * 0 * 11 * 1$. The 0's and 1's here represent pronouncements of what the bit value in that particular position must be, while the $*$'s represent bit positions where the value is unconstrained, a "wildcard" if you will. In our example, the template $001 * 0 * 11 * 1$ refers to binary strings of length $n = 10$, and there are $2^3 = 8$ strings of that length that "match" the template, e.g. 0010001101, 0011001101, 0010011101, etc. These 8 inputs will cause f to output 1, while any other strings of length 10 will cause f to output 0.

We'll use w to denote the number of wildcard slots in our function template, so the size of the support is 2^w. We can start with a natural variant of our subset-sum inspired approach for a point function: we will sample random values $s_{b,i} \in \mathbb{Z}_p$ for $b \in \{0,1\}$ and $i \in [n]$, with the constraint that the sums corresponding to inputs that match our pattern template all sum to 0, while other input combinations do not. This forces us to make values $s_{0,i} = s_{1,i}$ for all wildcard slots i. We will illustrate this by collapsing our notation to s_i for wildcard slots i. For our concrete example with template $001 * 0 * 11 * 1$, this would look like:

$$\begin{array}{cccccccccc} s_{0,1} & s_{0,2} & s_{0,3} & s_4 & s_{0,5} & s_6 & s_{0,7} & s_{0,8} & s_9 & s_{0,10} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_4 & s_{1,5} & s_6 & s_{1,7} & s_{1,8} & s_9 & s_{1,10} \end{array}$$

Here each s_i and $s_{b,i}$ represents a fresh and independent random variable, except for $s_{1,10}$, which is defined by the necessary relationship:

$$s_{0,1} + s_{0,2} + s_{1,3} + s_4 + s_{0,5} + s_6 + s_{1,7} + s_{1,8} + s_9 + s_{1,10} \equiv 0 \bmod p.$$

Evaluation of the function on a given input $x_1 \dots x_{10}$ is performed by computing:

$$\sum_{i=1}^{10} s_{x_i,i} \overset{?}{\equiv} 0 \bmod p.$$

In what sense might this be an obfuscation? Well, it may hide something about the constrained bit values that form the underlying structured sum, if there are enough constrained values to make spotting the sum difficult. However, it clearly does not hide the location of the wildcards in the pattern. And this is not something that can be fixed by moving all of these values into the exponent of a group of prime order p where discrete log and related problems are computationally hard:

$$\begin{array}{cccccccccc} g^{s_{0,1}} & g^{s_{0,2}} & g^{s_{0,3}} & g^{s_4} & g^{s_{0,5}} & g^{s_6} & g^{s_{0,7}} & g^{s_{0,8}} & g^{s_9} & g^{s_{0,10}} \\ g^{s_{1,1}} & g^{s_{1,2}} & g^{s_{1,3}} & g^{s_4} & g^{s_{1,5}} & g^{s_6} & g^{s_{1,7}} & g^{s_{1,8}} & g^{s_9} & g^{s_{1,10}} \end{array}$$

We still assume that the identity element of $G =< g >$ (i.e. g^0) can be efficiently recognized (which is needed for the final conclusion of evaluation), so equality testing of the repeated instances of g^{s_4}, for example, will give away the location of the wildcard slots in this instantiation too.

So if we want to hide the wildcard locations, what might we do? It's clear we need to put different values into those slots, even though those values are not supposed to affect the ultimate outcome of the function evaluation. One thing that can help us is that there's no reason we have to combine the pieces we select from each slot in a way that is agnostic to where we pulled them from. For example, we can compute a more general linear function,

$$\sum_{i=1}^{10} c_{x_i,i} s_{x_i,i} \overset{?}{\equiv} 0 \bmod p,$$

where the coefficients $c_{x_i,i}$ are allowed to vary with x_i and i. This gives us enough degrees of freedom to use, say, a linear secret-sharing scheme.

In particular, let's see what happens if we use Shamir secret sharing, and set the $s_{b,i}$ values to be evaluations of a secret polynomial on specified points in all bit positions and values that match the pattern, and fully random values elsewhere, e.g. for our pattern $001 * 0 * 11 * 1$:

$$\begin{array}{cccccccccc} P(1) & P(3) & r_3 & P(7) & P(9) & P(11) & r_5 & r_6 & P(17) & r_7 \\ r_1 & r_2 & P(6) & P(8) & r_4 & P(12) & P(14) & P(16) & P(18) & P(20) \end{array}$$

Here, P is a polynomial of degree 9 whose coefficients are chosen randomly over $\mathbb{Z}_p$, up to the constraint that the constant term is 0. In each position b, i where the bit b matches the pattern specification for the i^{th} position (which includes both bit values for wild slots), we place the value $P(2i - b - 1)$. In positions that do not match the pattern, we place fully random values r_k.

To evaluate the function on a particular input $x_1 \dots x_{10} \in \{0, 1\}^{10}$, we will use the value from each pair corresponding to the bit value x_i, and try to interpolate the polynomial P from our 10 values as if each is equal to $P(2i - x_i - 1)$. If we get 0 as the constant term, we output 1. Otherwise we output 0.

It is clear that this will output 1 for the desired support, and for any input that doesn't match the pattern, the answer will not be 0 with high probability. And at least in isolation, a pair of values like $P(7), P(8)$ in the first wildcard slot above will look like random values, and will not give away the wildcard positions.

But of course, this polynomial setup is also an error-coding code, and more than half of the values here are correct evaluations of the polynomial. So very quickly we have to worry about the prospect of unique and list-decoding of Reed-Solomon codes, which would reveal our wildcard locations if we fall within the reach of such techniques.

We are skating on the edge of a natural tension here: we want the helpful flexibility of secret sharing, in that different combinations of valid shares can be equivalently used to compute the desired outcome, but we do not want much (if any!) of the robustness that tends to come along for the ride, allowing us to distinguish valid shares from invalid ones. There is a whole subfield of secret sharing research that seeks to build "robust secret sharing schemes," maximizing this very property that for us is a blow to our hope of obfuscating the wild card positions. We want something like "non-robust" secret sharing. We had a more colorful name for it in our research meetings: "shitty secret sharing": secret sharing that falls apart catastrophically as soon as a small number of incorrect shares are introduced.

This is something that *can* be achieved by throwing our values into the exponent of a group:

$$\begin{array}{cccccccccc} g^{P(1)} & g^{P(3)} & g^{r_3} & g^{P(7)} & g^{P(9)} & g^{P(11)} & g^{r_5} & g^{r_6} & g^{P(17)} & g^{r_7} \\ g^{r_1} & g^{r_2} & g^{P(6)} & g^{P(8)} & g^{r_4} & g^{P(12)} & g^{P(14)} & g^{P(16)} & g^{P(18)} & g^{P(20)} \end{array}$$

To reason about the potential properties of this construction, the easiest way is to use the heuristic of a "generic group." This means that we think about $G =< g >$ as if it is a black box: the box allows you to multiply elements of the group, e.g. $g^a g^b = g^{a+b}$, and hence to raise group elements to known powers, e.g. $(g^a)^s = g^{as}$. It also allows you to identify g^0 as the identity element (and hence to recognize equality of any two acquired instances of the same group element), but that's it! So not only do we think of the discrete logarithm as being hard, but we imagine that any other problem that is not trivially solved by a sequence of basic group operations is also hard. For instance, in a generic group, given g, g^a, g^b, it is hard to compute g^{a^2+b}, or to distinguish g^{ab} from a random element, or to distinguish g^{a+b^3} from a random element, etc.

All of cryptography is built upon (conjectured) separations between what is easy and what is hard. The generic group heuristic is this concept on steroids (and then also on ecstasy): we assume everything that is not inferable to be easy through a polynomial number of standard group operations is therefore hard. Obviously, this isn't completely true for any concrete group G. But it is kind of truth-y for some groups G, at least enough that cryptosystems justified solely under the generic group heuristic have not historically suffered damaging attacks.

In such a framework, we immediately have a separation between the function evaluation that we want to be easy/efficient, since it is linear, and the decoding algorithms we want to be hard/hopelessly inefficient, which are non-linear. Ultimately what this gives us is that when the number of wildcard slots is not too high, we can argue that our obfuscated function is computationally indistinguishable from the distribution:

$$\begin{array}{cccccccccc} g^{r_1} & g^{r_3} & g^{r_5} & g^{r_7} & g^{r_9} & g^{r_{11}} & g^{r_{13}} & g^{r_{15}} & g^{r_{17}} & g^{r_{19}} \\ g^{r_2} & g^{r_4} & g^{r_6} & g^{r_8} & g^{r_{10}} & g^{r_{12}} & g^{r_{14}} & g^{r_{16}} & g^{r_{18}} & g^{r_{20}} \end{array}$$

where all of $r_1, \ldots, r_{20}$ are random, so this effectively represents a function with no support (with high probability). Since this distribution is independent of the pattern, we have hence ruled out any leakage of the placement of wildcard slots, as well as the bit values in the constrained positions.

We proved this extensively ourselves in [BKM$^+$18], though it was later pointed out to us that much of this follows as an application of the analysis of Peikert in [Pei06]. That is another kind of subplot that is often part of the research experience: the "I proved this all by myself because I didn't know some of it was already proved elsewhere, but so-help-me-god I'm going to find a reason to show you *my* proof anyway because I worked so hard on it, you know? and maybe it gets slightly better parameters in some way or something..." subplot. I mean, we do think our proof has independent value. I mean, really we do. I'm not just saying that. And our construction for the purposes of wildcard obfuscation is certainly novel. And in any case, Chris Peikert is a great cryptographer, so it's pretty good company to be in. Just saying. There are also follow-up works now that improve and expand upon our construction and analyses: [BLMZ]. And we are not defensive about any of this at all.

Chapter 8: The Postscript

Every great story is supposed to have a satisfying arc: it should have ups, it should have downs, it should tug at your heart strings, stomp on your dreams, and then magically restore them to an even greater glory at the last possible moment. When we embarked upon this work, we hoped we could reach an understanding more satisfying and more comprehensive than the place I leave you now.

But journeys are not about their endings. In research, unlike classic stories, endings are rare and typically either unsatisfying or sad. Beginnings are exciting and celebrated, but middles get a bad rap. We are at a middle point in the development of cryptographic obfuscation, a middle point in the development of lattice-based cryptography, a middle point in our understanding of provable security more generally, and a middle point in our understanding of how cryptography fits into the larger universe. And thank goodness! For we should be proud and happy for the progress we have made and the things we learned, and grateful for the joys of discoveries and failures yet ahead.

References

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, pages 1–18, 2001.

[BGK$^+$14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 221–238, 2014.

[BKM$^+$18] Allison Bishop, Lucas Kowalczyk, Tal Malkin, Valerio Pastro, Mariana Raykova, and Kevin Shi. A simple obfuscation scheme for pattern-matching with wildcards. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, pages 731–752, 2018.

[BLMZ] James Bartusek, Tancrède Lepoint, Fermi Ma, and Mark Zhandry. New techniques for obfuscating conjunctions. In *EUROCRYPT*, volume 2018.

[CGH$^+$15] Jean-Sébastien Coron, Craig Gentry, Shai Halevi, Tancrède Lepoint, Hemanta K. Maji, Eric Miles, Mariana Raykova, Amit Sahai, and Mehdi Tibouchi. Zeroizing without low-level zeroes: New MMAP attacks and their limitations. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 247–266, 2015.

[CGW15] Jie Chen, Romain Gay, and Hoeteck Wee. Improved dual system ABE in prime-order groups via predicate encodings. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 595–624, 2015.

[CHL$^+$15] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 3–12, 2015.

[CLLT16] Jean-Sébastien Coron, Moon Sung Lee, Tancrède Lepoint, and Mehdi Tibouchi. Cryptanalysis of GGH15 multilinear maps. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 607–628, 2016.

[CLLT17] Jean-Sébastien Coron, Moon Sung Lee, Tancrède Lepoint, and Mehdi Tibouchi. Zeroizing attacks on indistinguishability obfuscation over CLT13. In *Public-Key Cryptography - PKC 2017 - 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I*, pages 41–58, 2017.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 476–493, 2013.

[CM14] Melissa Chase and Sarah Meiklejohn. Déjà Q: using dual systems to revisit q-type assumptions. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 622–639, 2014.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. volume 22, pages 644–654, 1976.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 1–17, 2013.

[GGH$^+$13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49, 2013.

[GLSW15] Craig Gentry, Allison Bishop Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 151–170, 2015.

[GLW14] Craig Gentry, Allison Bishop Lewko, and Brent Waters. Witness encryption from instance independent assumptions. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 426–443, 2014.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92, 2013.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 20–31, 1988.

[Lew12] Allison Bishop Lewko. Tools for simulating features of composite order bilinear groups in the prime order setting. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 318–335, 2012.

[LPSS14] San Ling, Duong Hieu Phan, Damien Stehlé, and Ron Steinfeld. Hardness of k-lwe and applications in traitor tracing. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 315–334, 2014.

[MSZ16] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 629–658, 2016.

[OT10] Tatsuaki Okamoto and Katsuyuki Takashima. Fully secure functional encryption with general relations from the decisional linear assumption. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 191–208, 2010.

[Pei06] Chris Peikert. On error correction in the exponent. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 167–183, 2006.