Local Mixing Documentation

Nicholas Ho

1 Introduction

Indistinguishability Obfuscation (iO) is immensely powerful. For instance, it can be used to achieve almost every other cryptographic task, such as trapdoor permutations or non-interactive zero-knowledge. If combined with lossy encryption, even a beast like fully homomorphic encryption can be achieved.

The history of achieving iO is quite intriguing. In fact, there have already been results that prove iO can in fact exist. However, there are numerous problems with these existing tools. Some of these tools include evasive LWE, multi-linear maps, learning parity with noise, etc. These tools can be highly structured resulting in constructions that are both complex and impractical. As a result, local mixing serves to answer the following question: Can we achieve iO with "first principles"? The [CCMR'24] paper proposes a solution to such a question: local mixing.

This paper serves as a documentation for our experiments, results, as well as questions that remain for local mixing. We note that we solely rely on the structure of reversible circuits, but this of course is not a problem as we can reduce any arbitrary circuit to a reversible circuit.

2 Strategies

The below strategies will all follow the same ideas. Namely, we want to make simple actions and use minimal structure in order to obfuscate our circuits. In the pursuit of simplicity, our circuits will thus only contain gate r57. Standard gates like NOT, AND, OR, NOR, NAND, etc, can all be represented by gate r57. Thus, this simplification does not impede us from our goal of general-purpose obfuscation.

Definition 2.0.1. *Let g be gate r57 on inputs A, B, and C, with A being the active wire, B being the first control pin, and C being the second control pin. Then $A = B \wedge \neg C$.*

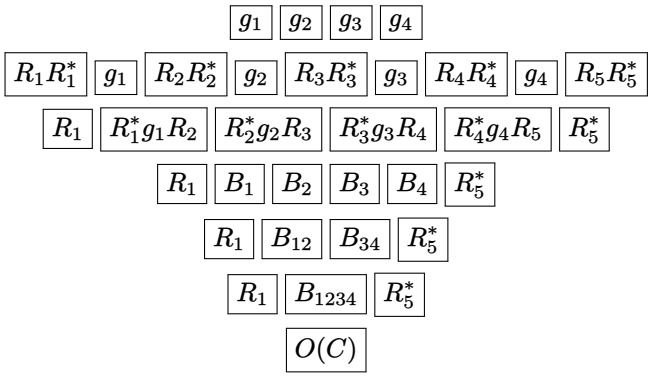
A good question to answer is what type of actions we are allowed to do to follow the rule of "first principles". The basic local mixing strategy is thus to only allow small local and equivalent perturbations of a circuit in the hopes that with enough of them, we can get a global effect. In other words, given a circuit, we sample and replace many subcircuits of equivalent functionality. Of course, there are many ways to do this that do not actually incur a large enough effect. For example, say we have a circuit on 64 wires with 1000 gates and only make 100 replacements on 6 gate subcircuits. Then, with most replacement methods, an observer can merely find these points of replacements and potentially even undo them. We then have a number of concerns that we will need to address. Firstly, how can we sample replacements? This is a question that we did not think too deeply about at the beginning of the project, and so will be more deeply talked about later. Secondly, how can we mix gates in a way that we can not identify points of replacement?

With these questions in mind, let us introduce the obfuscation strategies that we have tried.

1. **The Two Phase Strategy**: This strategy splits up the obfuscation process into two phases. The first phase is the *inflationary phase*. In this phase, we sample random subcircuits and make replacements

that contain more gates than the subcircuit we are replacing. Thus, we slowly inflate our subcircuit. We then move on to the *kneading phase*. In this phase, we sample random subcircuits and make replacements with equal length random circuits in the hopes that we can spread out the randomness that we injected into our circuit in the inflationary phase.

2. **The Butterfly Method**: The butterfly method utilizes compression, something that we will talk more about later. As opposed to a two stage method, the butterfly method has many stages of (inflationary $\rightarrow$ compression). Suppose we have a circuit C. We then wrap each gate with an identity of the form $R_i R_i^{-1}$, a simple identity. We can sample R_i completely randomly and so we have a "stupid" identity. Afterwards, we group up each $R_i^* g_i R_{i+1}$ and randomize it it to get B_i. This can be done by attempting to compress it, expand it, or make equal length replacements of equivalent functionality. All that remains is to then merge the B_i together, compressing as we do so. The diagram below shows one round of this in more detail. Ideally, we would use enough rounds for this to effectively "mix" our gates.



3. **Pair Replacement Methods**: The pair replacement methods are different strategies that utilize one type of replacement: the pair replacement. Given two gates g_i and g_{i+1}, we can replace them with some identity I_i by relabeling the first two gates to be equal to the pair. In other words, we have $I_i = g_i g_{i+1} B_i$, which gives us $B_i^* = g_i g_{i+1}$. These strategies will make repeated replacements of this type before moving onto a compression phase. These two phases will then repeat. This is meant to be seen as a smaller version of the butterfly method.

2.1 Sampling Subcircuits

In the above strategies, it is important for us to sample random subcircuits and then replace them. We will discuss how to make these replacements in a later section. Before we talk about our methods, it is useful to define collisions between gates.

Definition 2.1.1. *Two gates $g_1 = (A_1, B_1, C_1)$ and $g_2 = (A_2, B_2, C_2)$ collide if one of the following holds:*

1. $A_1 == A_2$
2. $A_1 == B_2$
3. $A_1 == C_2$
4. $B_1 == A_2$
5. $C_1 == A_2$

The first obvious way to sample subcircuits is by sampling a random contiguous subcircuit within our circuit. If we view our circuit as an array of gates, then this is merely taking a window of some size. This is, of

course, very easy to do.

The other method is to take "convex subcircuits". The meaning of convex here directly lies in finding convex subgraphs of a directed graph. In order to turn a circuit into its graph representation, let each gate be a node i and for every other gate j that collides with i, is such that $i < j$, and there is no k with $i < k < j$ that already collides with i, then there is a directed edge from i to j. We do this for all nodes in order to get our skeleton graph. From this, we can sample a convex subgraph and then take the circuit version of it in order to get what we call a **convex subcircuit**. As we have convexity, we can then rearrange gates in the original circuit to make this convex subcircuit a contiguous subcircuit. As a reminder, we note that gates can be swapped without changing the functionality of the original circuit if they do not collide. Unfortunately, the algorithm for finding convex subcircuits is slow when used hundreds of millions of times. This is our largest bottleneck when it comes to speed.

2.2 How we measure success

As we discuss our strategies, it will of course be important for us to measure how well we are doing. For that, we have two methods: heatmaps and incompressibility.

Firstly, heatmaps can be computed by taking circuits C_1 and C_2, and then computing a state after every gate in each respective circuit. We can then take the hamming distance of the two resulting states at each point in the circuit and with some normalizing across all wires to get the hamming distance to be between 0 and 1, we can get a heatmap between two circuits. We use *green* to denote "random" and *red* to denote "equal", however, as we will see later, our older heatmaps have *purple* and *green* instead. As an idea of how these heatmaps should look, two equal heatmaps should have a red diagonal line along $y = x$. On the other hand, two completely random circuits should have a completely green heatmap. Two circuits that are random but have the same functionality should have red in the bottom left and the top right corners, since these depict 0 gates and all gates, but then green everywhere else. Below is a sketch of this.

Figure 1: Sketch of an ideal heatmap

Our other measure is incompressibility. In other words, how well can we sample subcircuits and find replacements with a smaller number of gates? In order to understand this problem better, we need to talk more about how we are making our replacements in the first place. We will talk more about compression in a later section, as our initial strategy, the two phase strategy, does not use compression.

3 The Two Phase Strategy

The two phase strategy was the initial method that we used in order to try and achieve local mixing. For more details on it, see the [CCMR'24] paper. However, here, we will provide some results.

While we were experimenting with this method, we did not use compression nor heatmaps in order to determine how well we were doing. However, the loose proof given in [CCMR'24] was enough motivation to believe that this method could work well. We tested numerous things here and got many different results. We remind that the heatmaps here have an outdated coloring scheme, with *purple* meaning the two circuits appear random to each other and *green* to mean the two circuits are equal.

Firstly, our results here are on 64 wires. One essential thing for us to figure out, as long as we assume that the two phase method works, is the correct parameters to choose. We need to answer how much we need to inflate, as well as how many rounds of kneading is necessary. The principal idea behind the inflationary phase is to first add padding that allows room for our circuit to be randomized. Afterwards, it is the kneading that stops attackers from undoing our replacements.

When it comes to making our replacements in each phase, the easiest method would be to simply precompute a table of all possible circuits of a given n wires and m gates. For now, this is what we will be moving forward with.

3.1 The $10,000 Bounty

Trusting that the inflationary and kneading phases would be enough to stop any attacks on our circuit, we started up a bounty. We started with a completely random circuit with 64 wires and 1014 gates. Afterwards, we repeated the inflationary phase until we reached 12,111 gates and then did hundreds of thousands of rounds of the kneading phase. If the kneading phase was done correctly, then all replacements in the replacement phase should be irreversible. The goal of the bounty is to find an equivalent circuit on 1014 gates or less, essentially finding the original circuit and showing that the obfuscation was ineffective. The bounty can be found here.

The bounty was quickly broken by Killari. There were a number of techniques that were used. One of them, was to use a precomputed rainbow table, much like we did to make our own replacements. However, Killari would try to make sampled subcircuits smaller. In other words, by sampling random subcircuits, Killari was able to compress the circuit back down. In fact, the circuit was compressed to less than 1000 gates. This is when it became obvious that we needed to take strongly consider compression attacks. For more details, see Killari's blog post on breaking the bounty A detailed discussion can be found here.

3.2 After the Bounty

Now that we knew that our two phase method was incomplete, we moved on to looking at heatmaps. We wanted to better understand the circuits that we could "obfuscate" with this method. As a result, we generated numerous heatmaps to understand what was happening with different changes in parameters. Namely, how the circuits look after more rounds of kneading.

Keeping to 64 wires, it was clear that a couple thousand number of rounds for the kneading stage was insufficient. The heatmap below shows that there is still an extremely clear correlation between the original circuit and the "obfuscated" circuit as there is a clear green line across $y = x$.

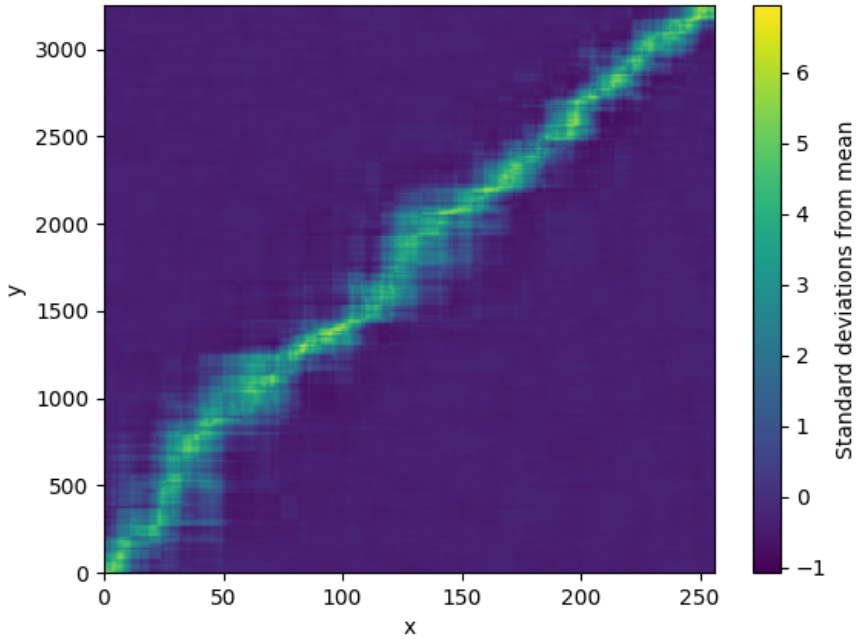


Figure 2: Heatmap showing the two phase method with 1,000 rounds of kneading

The two heatmaps now show what the same circuit heatmaps look like with 100,000 rounds of kneading and 1,000,000 rounds of kneading.

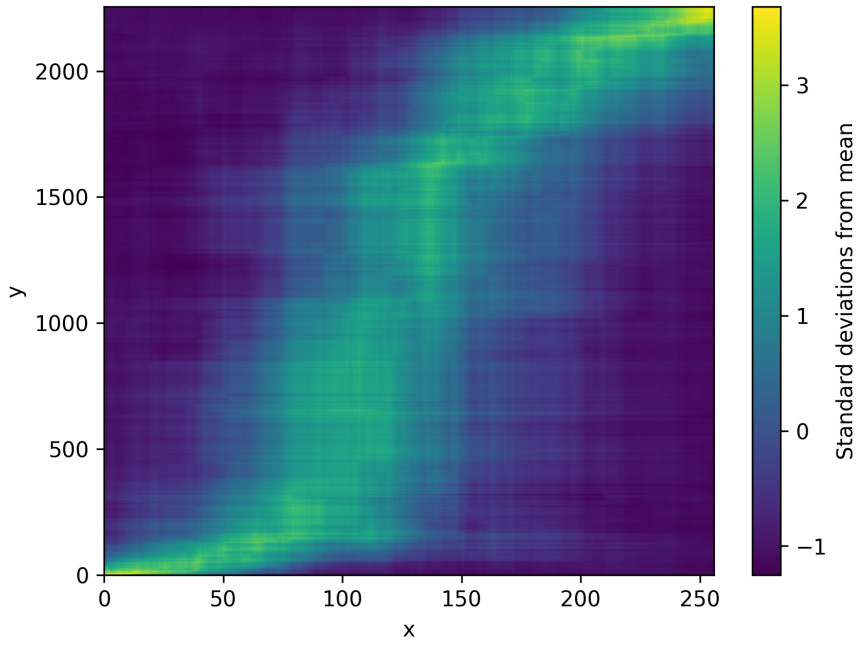


Figure 3: Heatmap showing the two phase method with 100,000 rounds of kneading

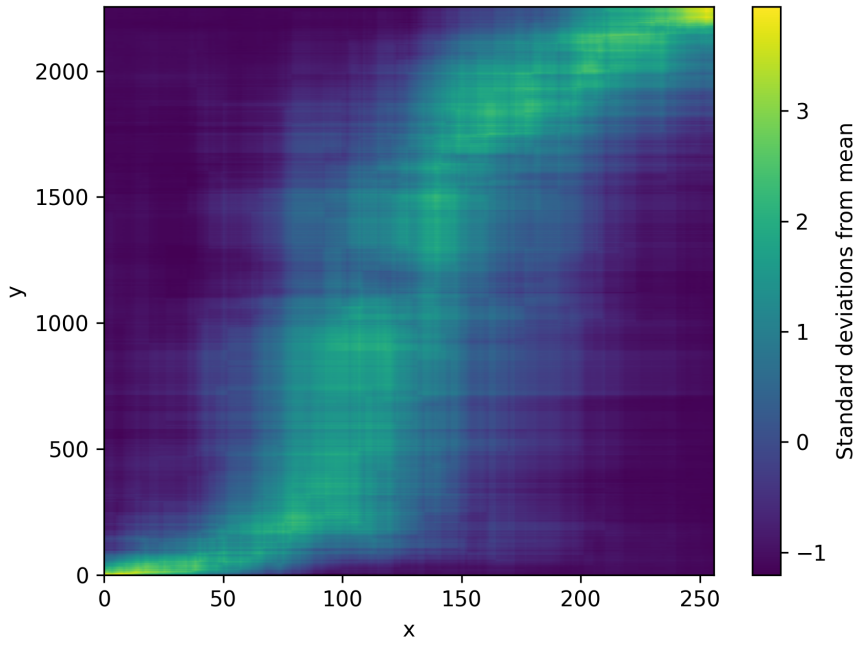


Figure 4: Heatmap showing the two phase method with 1,000,000 rounds of kneading

While it is clear that our heatmap is indeed "spreading out", which means that we are indeed seeing more randomness, we also see that it would require tens or even hundreds of millions of kneading rounds. Thus, it was obvious that something needed to be changed. Below is a table for any circuit-wiring language that we use.

Table 1: Wire Index Encoding Used by Circuits

Wire range	**Encoded as**	**Notes**
0–9	`0-9`	Decimal digits
10–35	`a-z`	Lowercase letters
36–61	`A-Z`	Uppercase letters
62–71	`! @ # $ % ^ & * ( )`	Special symbols (set 1)
72–82	`- _ = + [ ] { } < > ?`	Special symbols (set 2)
≥ 83	`~`^{k}`X`	k tildes followed by base-83 digit

4 Random Circuits

We mentioned before that one easy way to sample random circuits, is to use a precomputed table of circuits. However, this does not take into account circuits that are essentially the exact same circuit, but differ in trivial ways. For instance, the below circuits are all equivalent circuits

$$123; 456; 789$$

$$123; 789; 456$$

$$789; 123; 456$$

Noticing that none of these gates even collide shows that we can actually order the gates in any way that we want. This means that many of our replacements in the kneading stage, were just trivial replacements as seen here.This gives us a new problem of defining circuit equivalences beyond trivial gate swaps.

4.1 Canonicalization

There are two types of trivialities we can identify. First, it is the one described in the last section where we can swap non-commuting gates. For the second, let us first consider the following two circuits.

$$123; 345$$

$$456; 678$$

We can see that the gates now collide and so we can not just swap things around to make them equivalent. In fact, they aren't even equivalent at all. However, the structure that they have is identical. Namely, there is one collision between the second control pin of the first gate and the active pin of the second gate. We can see that mapping the wires

$$1 \to 4$$
$$2 \to 5$$
$$3 \to 6$$
$$4 \to 7$$
$$5 \to 8$$

allows us to map the first circuit to the second one. Thus, while the circuits are effectively different circuits, we can see that they are extremely similar. By considering only one of these circuits in our rainbow table of possible circuits, our random sampling is only amongst truly different and random looking circuits.

In order to find the second type of similarity, we can just look at the permutations of the circuits and define a canonicalization on the permutations. This gives us two types of canonicalizations for us to consider: circuit canonicalization and permutation canonicalization.

It is important for our canonicalization algorithms to be efficient as we will be constantly computing them. One extremely simple algorithm for circuit canonicalization is by just taking the lexicographic ordering, with respect to colliding gates. This is very efficient already as it is essentially just a sorting algorithm. On the other hand, computing permutations is much harder. While we know that there exists a poly-time algorithm, we have not yet found this. Thus, the best we can usually do is brute forcing every possible bit shuffle in order to find the lowest lexicographical ordering. It remains an open problem to find a better algorithm for this and this is one of our main bottlenecks. Once the bit shuffle that corresponds to the smallest permutation lexicographically, we can just apply the bit shuffle to the circuit and of course undo the shuffle whenever we want so that we don't lose any circuits in this canonicalization, just like how we can merely shuffle gates around again to "undo" circuit canonicalization.

4.2 Rainbow Tables

Using precomputed rainbow tables was the main method that we used in order to sample random circuits, as we could simply look up a circuit's functionality in the rainbow table and then have it spit out something equivalent. With our new notions of canonicalizations, we can greatly decrease the number of possibilities stored in our rainbow tables, as we saw that many circuits are already equivalent. It turns out that this means that many permutations only have a single circuit that corresponds to it. It is actually hard to find many circuits that have many "friends", where two circuits are friends if they differ in both canonicalizations above and still have the same permutation (which corresponds to the same functionality).

There are two main things to now consider: storage and speed. Storage is greatly helped by canonicalizations. Let n be the number of wires and m be the number of gates. Let us consider the rainbow table for $n = 5$ and $m = 5$. This has $n(n-1)(n-2)^m = 777,600,000$ total circuits before canonicalizations. After canonicalization, we can bring thus number down to about 2.5 million circuits.

We use a mix of SQL and B-trees (LMDB) in order to store these circuits. SQL is nicer than pure binaries as we can easily search in them automatically, without needing to convert the data in the binary files. However, as the number of circuits grows exponentially in both n and m, SQL is extremely slow for large n, m, even with indexing. Thus, using a B-tree like LMDB, which has fast reads, we can circumvent a lot of this. However, with the slow writing of LMDB, we were unable to store tables like $n6m5$ nor $n7m4$ and must use SQL for these. For all the table we have stored, we have $n3m1$, ..., $n3m10$, $n4m1$, ..., $n4m6$, $n5m1$, ..., $n5m5$, $n6m1$, ..., $n6m5$, $n7m1$, ..., $n7m4$. To store them all in SQL without permutation canonicalizations takes over 500 GB. The store it all in LMDB, once with both canonicalizations, once with only circuit canonicalization, and then a second table to keep track of friends, we only need around 5 GB. Thus, for most of our use cases, we stick to LMDB.

We briefly talked about speed above, but we will repeat it here for completeness. SQL is much slower than LMDB, so when we can, we stick to LMDB. On the other hand, using a brute force method to compute permutation canonicalizations is very slow for larger n. Thus, if we store only circuits up to circuit canonicalization, we can use our tables as a lookup for canonicalizations. So while using the canonicalization algorithm remains faster than a table lookup for $n = 3$, for all other tables, it faster or equivalent to use a lookup into LMDB. For $n = 4$, the times are quite similar, differing maybe by seconds for 10 million lookups.

On the other hand, for $n \geq 5$, the speed increase can be anywhere between 10x to 1,000x. This, while is helps a lot with our bottleneck of computing canonicalizations, does not solve it.

4.3 Compression

As we talked about before, we realized that it is important to not only consider heatmaps, but to also consider compression attacks. After all, much of the power of an attacker against general circuit obfuscation is the fact that they can freely sample subcircuits and replace them slowly. This brings us to two goals, motivated by both the heatmap and the compression attacks. Our first goal, is to make our circuits look random, which can be observed by looking at heatmaps. Our second goal is to make our circuits incompressible, or rather incompressible past a certain point. For instance, if we start with a 100 gate circuit and blow it up to 100,000 gates, then we don't mind if they can compress it down to 10,000 gates, as long as they don't get back down to 100 gates.

This serves as our motivation for the butterfly methods. We can also use our new rainbow tables in order to act as an attacker on our circuits. This changes our model of having two different phases to having many phases: inflate $\rightarrow$ compress $\rightarrow$ inflate $\rightarrow$ compress $\rightarrow$ inflate $\rightarrow$ compress $\rightarrow$ ...

We now state some things about compressibility. Stupid identities such as $214; 251; 123; 123; 251; 214$, created by just reversing an identity and then concatenating them, is extremely easy to compress. In addition, it is easy to compress identities created by combining friends from the rainbow table. However, it is extremely hard to compress completely random circuits. These facts are important to consider when looking at a new strategy for local mixing.

5 The Butterfly Methods

The main motivation for these methods, as mentioned above, is to get incompressibility. There are two versions of the butterfly method that we will talk about. Namely, they are the asymmetric butterfly method and the symmetric butterfly method. The asymmetric butterfly method is just the butterfly method but with each R_i being completely random. If we look at the blocks in asymmetric butterfly, they are of the form $R_i^* g R_{i+1}$. As the left and right circuits are random to each other, then this block is essentially completely random. As mentioned before, this makes the blocks extremely incompressible. On the other hand, the symmetric butterfly method is with each R_i being equal. While the symmetric butterfly method gives us much less randomness, the additional structure of each block $R^* g R$ makes them more compressible.

We note that $5OA$ will represent 5 rounds of symmetric butterfly on circuit A, and $5QA$ would be the same but with asymmetric butterfly.

We first test our experiments on a small number of wires, as we know that we can simply take contiguous subcircuits without sampling subcircuits with more wires than we are able to compress with our rainbow tables. Thus, we test on 5 to 7 wire circuits. We did many of our tests on the two identities, circuit A, a 55 gate identity on 5 wires, and circuit B, a 22 gate identity. We are interested in numerous heatmaps here, such as A vs OA, B vs OB, A vs B, and OA vs OB. The first two gives us an idea of how well we obfuscate in relation to the original circuit, while the last two give us an idea of how well we obfuscate equivalent, yet different circuits.

There was a problem we quickly ran into though. The butterfly methods were ineffective on a small number of wires. Even with many rounds of the butterfly methods, we would always be able to compress back down to the original circuit. If we started with an identity, we would always be able to compress back down to zero gates. This means that we are largely unable to compress blocks and the gates g in $R^* g R$ are not effectively being hidden. This means that we may need more "room" for the gates, which means we may need to use

more wires. We of course can still start with the 5 wire circuits, but the R that we sample should be over 64 wires. This would allow us to sample circuits from 3 wires to 7 wires, giving us much more freedom.

5.1 Moving on to more wires

One change becomes immediate as we move onto more wires. We can no longer sample contiguous circuits of size greater than 2 gates. This is because if two gates share no wires, then 2 gates will already span 6 wires and 3 gates will span 9 wires, exceeding the limitations of our rainbow table.

Before showing off some heatmaps, it is important to note that these results are from when we had a weaker compressor. So while the results won't match any longer, it maps our thought process as we changed our algorithms.

Below gives our first heatmaps of A vs B, A vs OA, A vs $2OA$. The results for B are very much the same and so will not be shown.

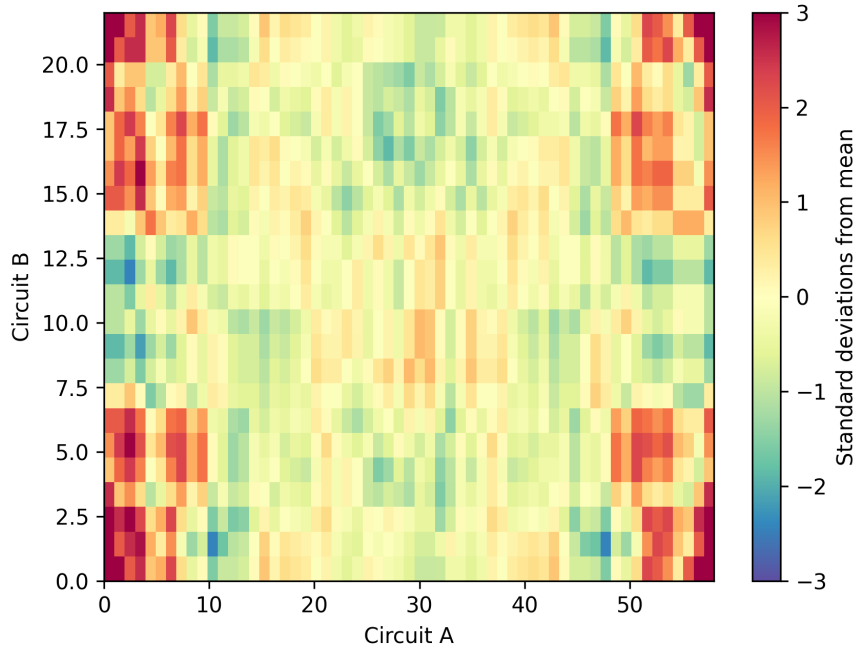


Figure 5: Heatmap showing the circuit A vs circuit B

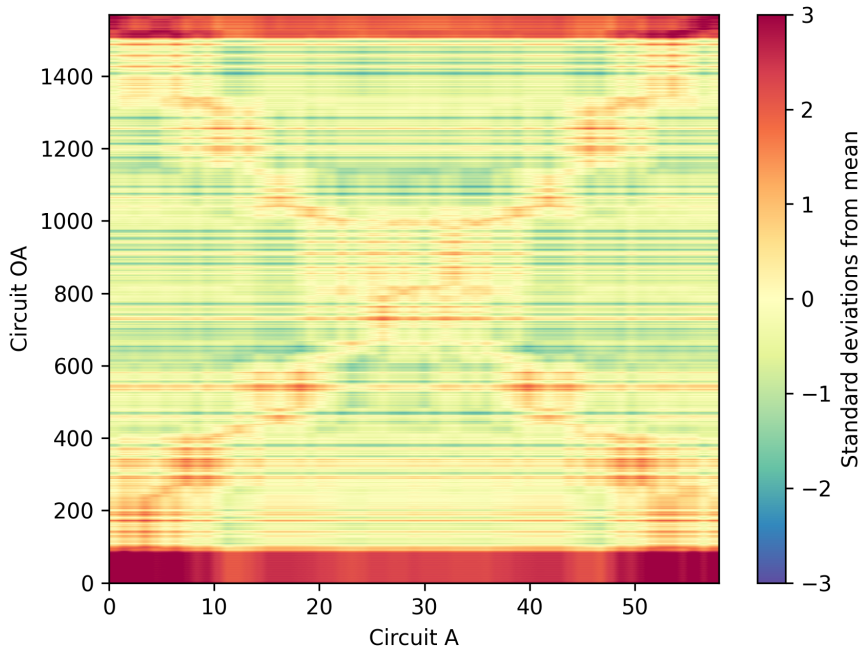


Figure 6: Heatmap showing the circuit A vs circuit OA

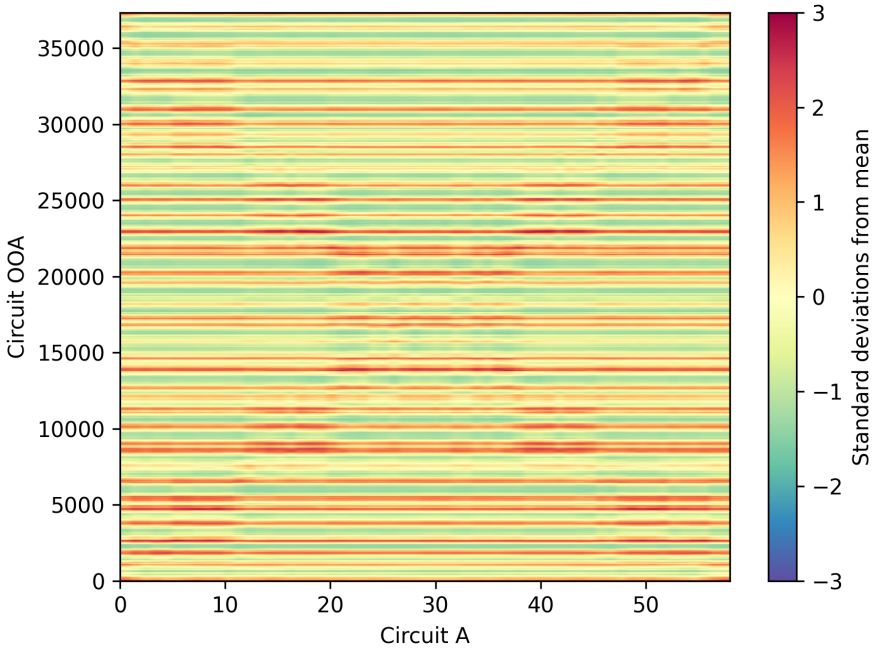


Figure 7: Heatmap showing the circuit A vs circuit $2OA$

From the above heatmaps, two things are clear. Firstly, there remains an obvious diagonal. Secondly, the number of gates has blown up greatly. The number of gates in each R is randomized between 6-25 gates, and with the weak compressor, it struggled to compress down at all, resulting in a large blow up in gates. However, there is still an obvious diagonal. In addition, due to runtime constraints, we would be unable to run another round as the number of gates would blow up even more. The final nail in the coffin, is if were to look at heatmaps on raw hamming distance data, rather than standard deviations. Standard deviations are useful to look at as they enlarge small differences for us to see, but when we look at the raw data, it tells a much more somber story.

The below are heatmaps where R, which we will now call *wings* is small (5-15 gates), medium (30-70 gates), large (100-150 gates) and large x2 (200-300 gates). We see that there is a gradual decrease in redness, which shows that indeed, longer wings will create more randomized obfuscated circuits. This is obvious as the gates we are trying to hide in each block are more "buried" within the arbitrary gates in the wings.

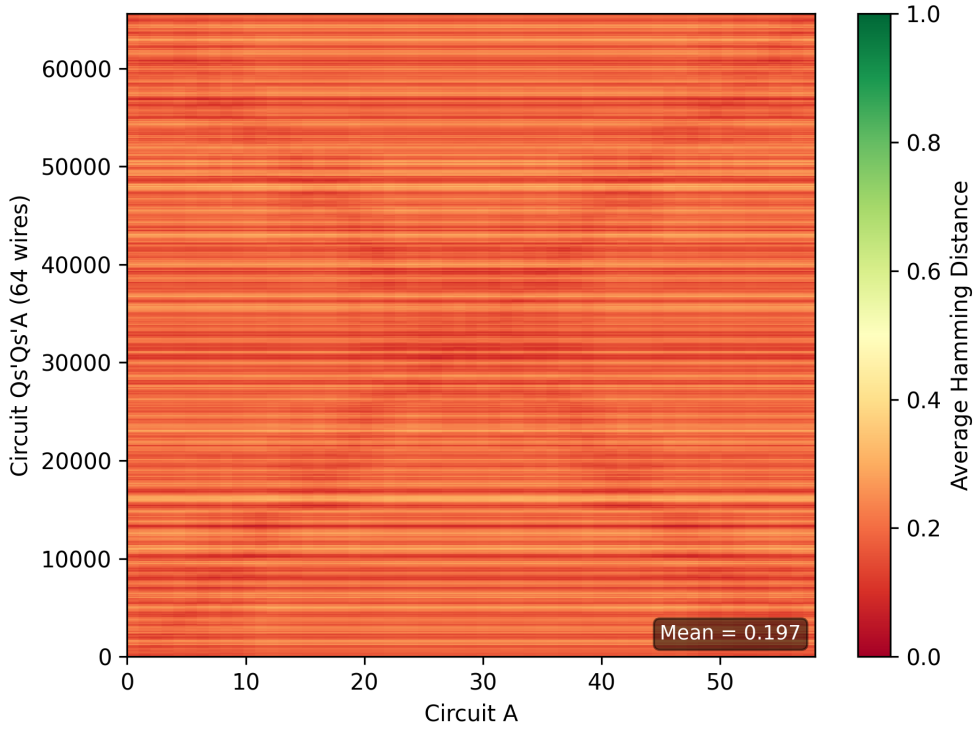


Figure 8: Heatmap showing the circuit A obfuscated with small wings

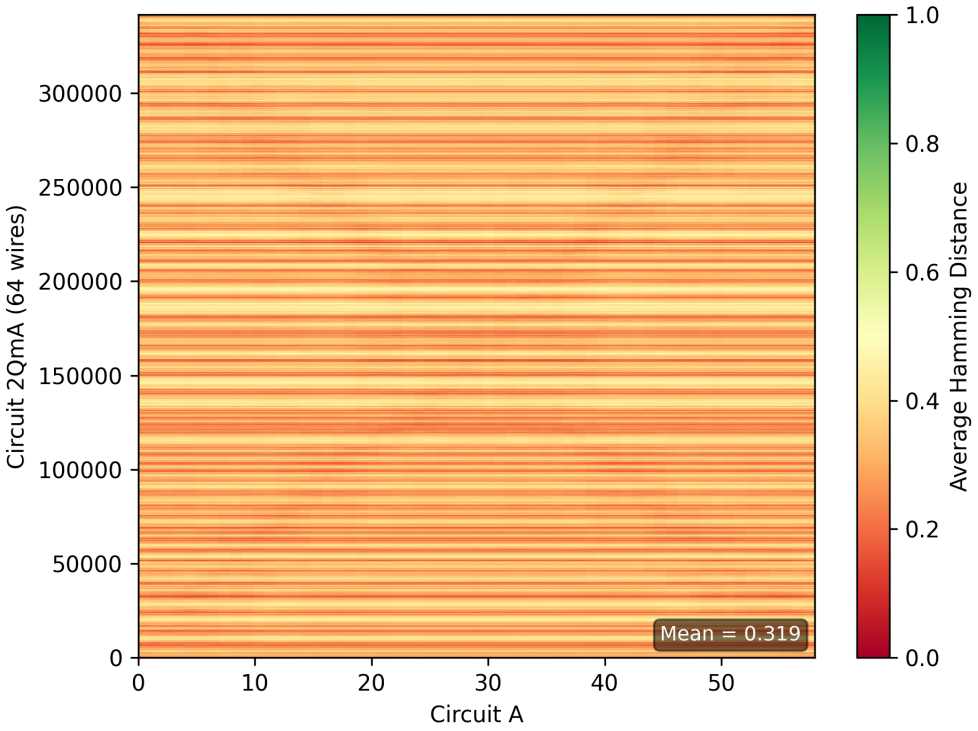


Figure 9: Heatmap showing the circuit A obfuscated with medium wings

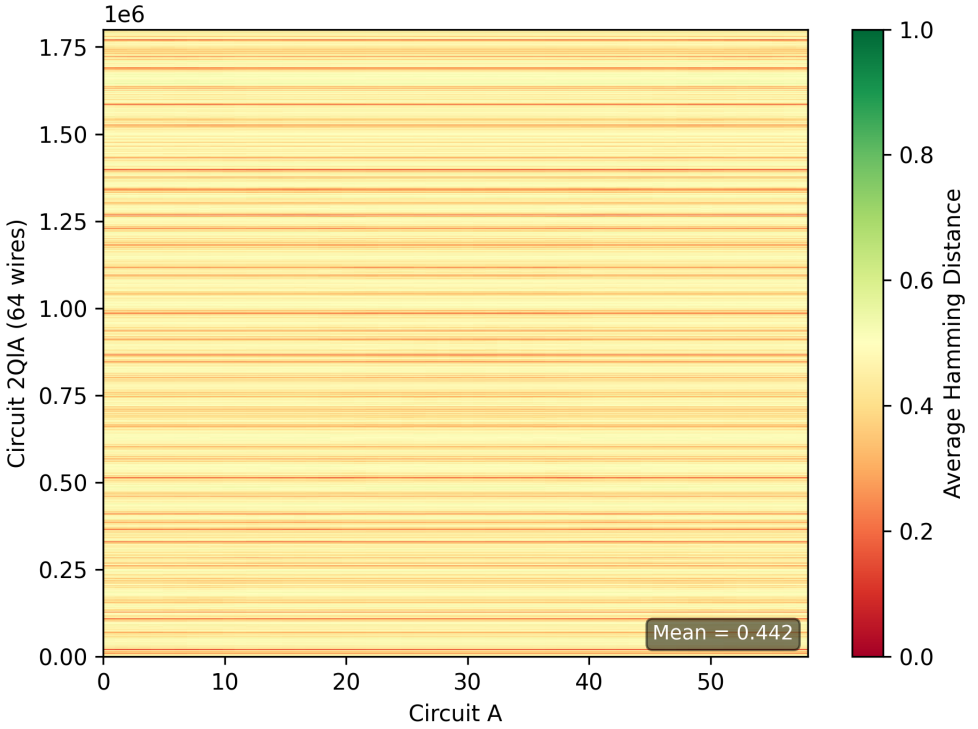


Figure 10: Heatmap showing the circuit A obfuscated with large wings

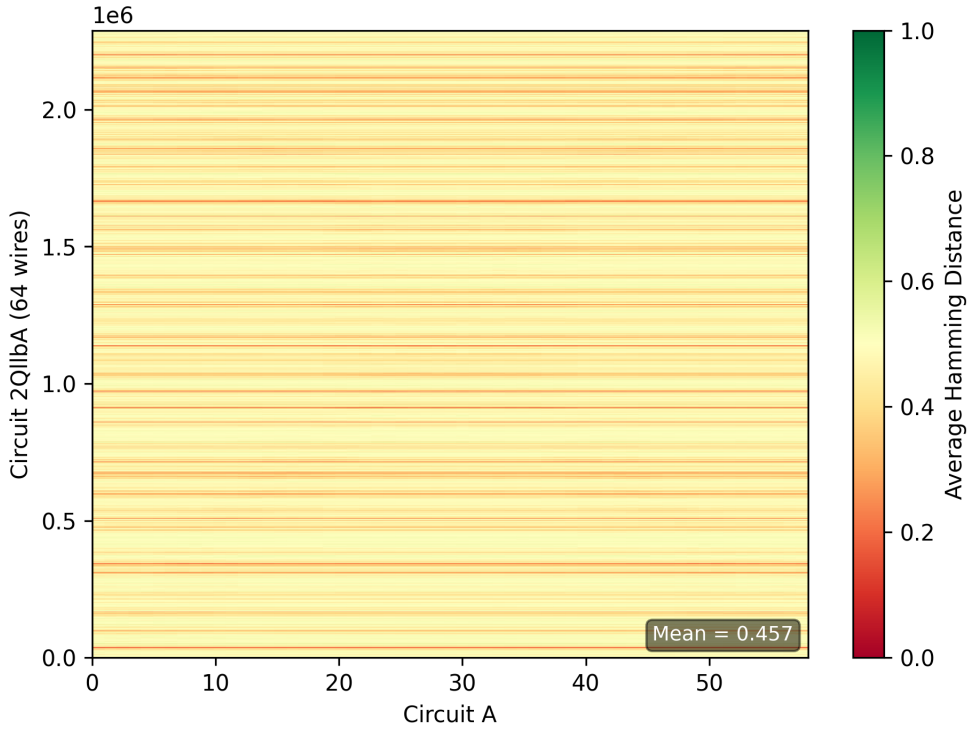


Figure 11: Heatmap showing the circuit A obfuscated with very large wings

With the above results, we kept with 100-200 gate wings and continued testing. It is at this point, that we made some improvements in speed to our compressor, allowing us to compress at a much faster rate, and much more. This improvement makes all the above results prior to this impossible. This actually made the entire butterfly method, as it was, useless. Suppose we have A and attempted to generate OA. Well this would compress completely and we would just have $A = OA$. If we then tried to get $2OA$, then this would be equivalent to as if we were trying to generate OA A. In other words, $A = OA = 2OA = \dots$. One thing that would be useful, is understanding at what point during compression our circuits stop looking random. The heatmaps below answer this question, to an extent. We start with a completely random circuit on 64 wires and 500 gates, and then take heatmaps as it is being compressed.

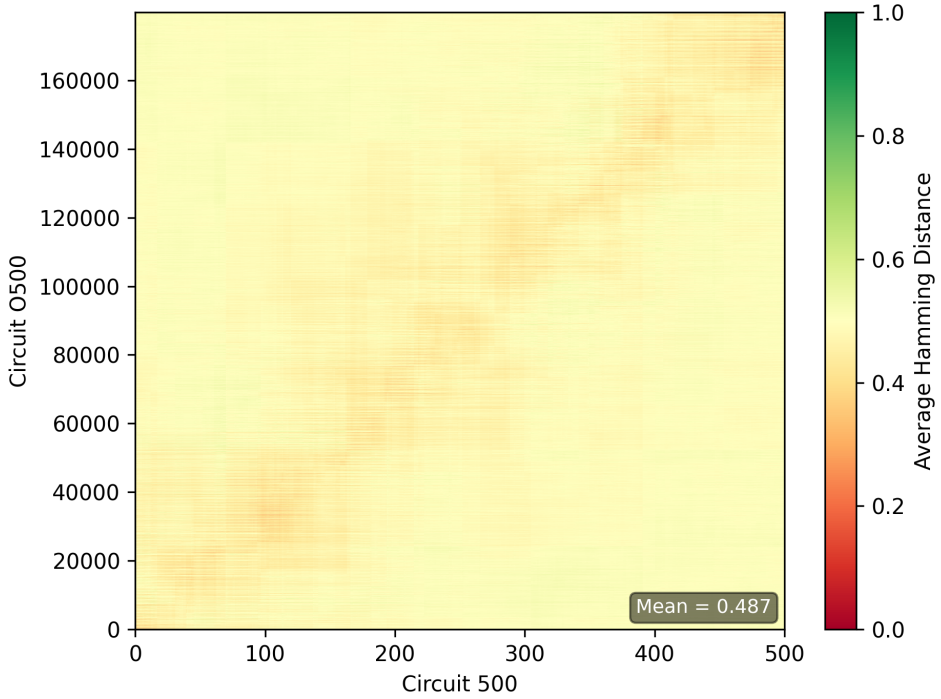


Figure 12: Heatmap showing circuit 500 before compression: 170,000 gates

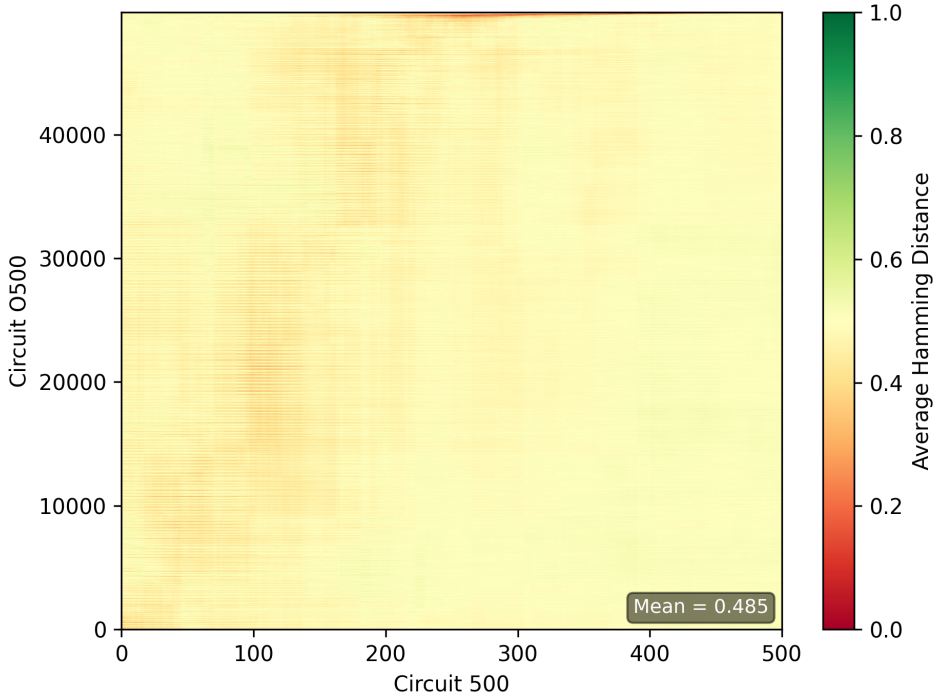


Figure 13: Heatmap showing circuit 500 during compression: 50,000 gates

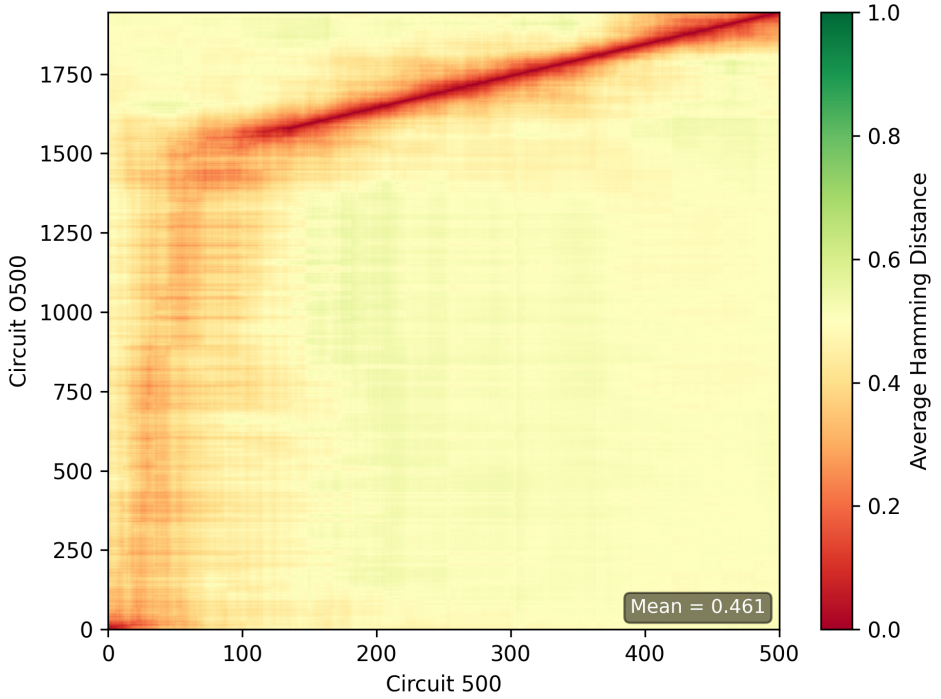


Figure 14: Heatmap showing circuit 500 during compression: 1900 gates

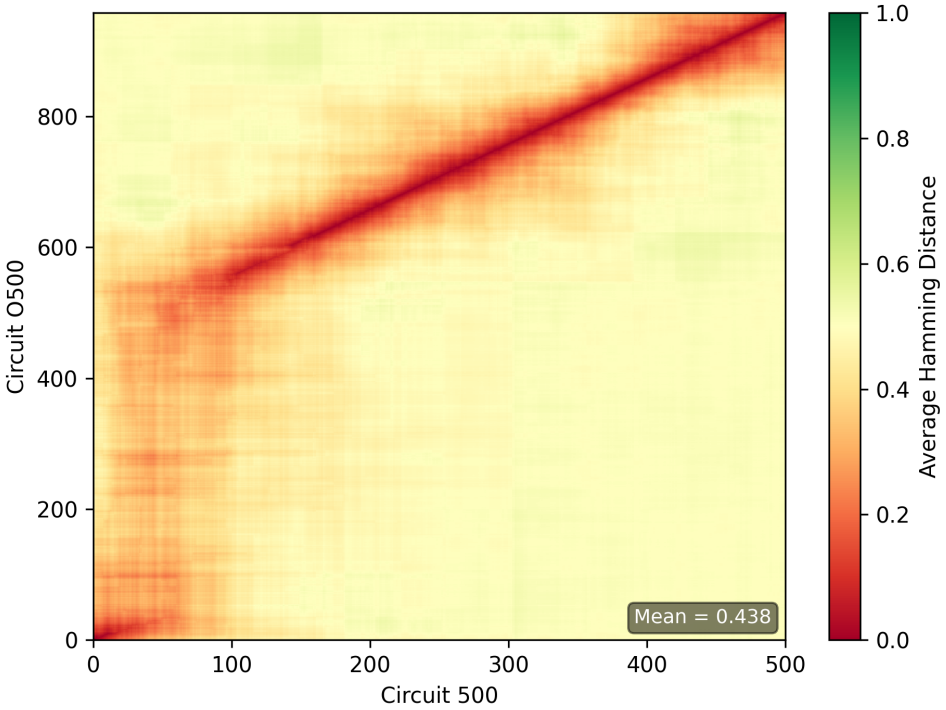


Figure 15: Heatmap showing circuit 500 after compression: 900 gates

The above results led us to instead stopping compression early, at around 1,000 gates. After each round, we would increase this by another thousand. So for the 10th round of compression, we would stop compression at 10,000 gates. It was only in the final round would we attempt to completely compress a circuit. This finally gave us incompressibility, without the extreme blowup in the number of gates. However, these heatmaps did not look the most random. The bigger problem was that if we changed our compressor to allow ancilla wires, which means when we sample a 5 wire subcircuit, we allow the compressor to look into the 6 or 7 wire rainbow table, we could actually compress these down completely. Thus, we needed to include ancilla wires to our randomization in some way.

5.2 Ancilla Wire Replacements

When we are making replacements, we would add randomization in two ways. The first way is by randomly sampling a subcircuit. The second way is by randomly choosing how many wires our subcircuit could be on. For example, we may try to only sample a 5 wire subcircuit as opposed to the much easier 7 wire one. One change that we could make to this, is to sample a 5 wire subcircuit, but then use the 7 wire rainbow tables to make the replacement. In other words, we are adding ancilla wires to our replacements. This alone, without anything above, gave us incompressibility. One useful thing about this process, is that it gave us even smaller incompressible circuits. Below, a heatmap shows that we still have the red beam in the top right corner, which tells us that we still need more randomization.

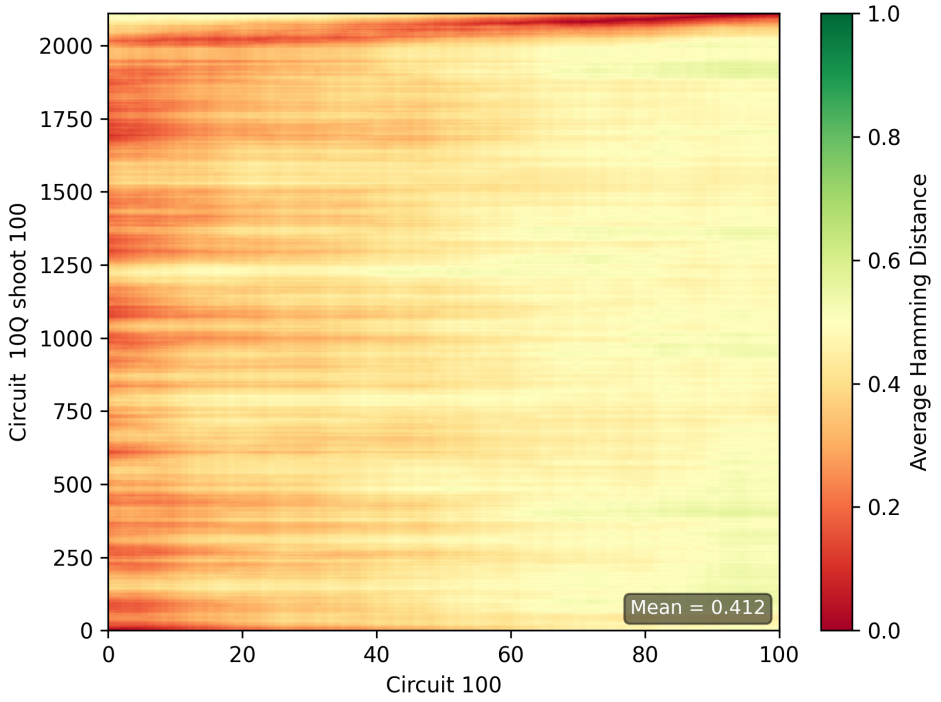


Figure 16: Heatmap showing a 100 gate circuit on 64 wires after the butterfly method with ancilla replacements

6 Blurring

With our new goal of randomization, we will consider various methods. As we are trying to make the heatmaps look more random, this means that we want to make our heatmap look more "blurry". Hence, we

call this section "blurring". One easy way to blur is to randomize the gate ordering of non-colliding gates and then "locking them in" via the incompressibility of the butterfly method. Another way to do so is to replace gates entirely with rewired identities. For instance, if we have g and an identity I, and then shuffle the bits of I such that the first gate of I matches with g, then we have $I = gB$, for some B. This means that $B = g$ and so we can replace g with B. These two ideas will be the pillars of our discussions moving forward.

6.1 Random Shooting vs Random Walking

This is our first, and more basic, method to blurring. The processes are described below.

Random Shooting:

1. Select a random gate g.
2. Select a random direction d, left or right.
3. Send g all the way d until it meets a gate that it collides with.
4. Repeat.

Random Walking:

1. Create the skeleton tree of the circuit
2. If any nodes have no parents in candidates, then add them to candidates (for the first iteration this is all nodes at level 0)
3. Select a random node from candidates, add it to c, and remove from candidates (nodes can not be re-added).
4. Repeat from step 2 until candidates is empty and all gates have been added to c.

The basic idea of these two methods is the same: to randomize the ordering of gates. The first one is extremely simple. There is no problem with sending a gate all the way left or right, because any gates in the same level can still be sent past it. For instance, say we have a circuit abc such that no gates collide. If we send c all the way to the left, then we have cab. We do not have to worry about c getting "stuck" there, as if we send b to the left, then we have bca. However, there remains a concern that the very last gate that we shoot will be at the end, or the farthest it can be. It is unclear whether this causes any damage or not. Random walking is meant to circumvent this problem as we are truly randomizing the nodes in each level of the skeleton tree. However, this is much more algorithmically complex as we need to convert our circuit to the skeleton graph. With a different data structure, it is possible that this is more efficient, but for now, we choose to use random shooting.

To further convince that this choice is not wrong, we have three tests to compare the two. For the first test, let circuit N be a circuit with no colliding gates on 64 wires. This circuit has 62 gates. There are three ways we can randomize this and retain functionality. First, we can just randomize all the gates with no algorithm. This essentially serves as our goal for the next two. Second and third are our proposed random shooting and random walking. As we are only on 62 gates, there is high variance, which means that heatmaps won't be the best way to view the results. Instead, we can record the mean hamming distance across hundreds of random states between N and the randomized circuits, and then randomize them 100 times to get a new mean each time. Taking the average gives us a better estimate on how we are doing.

For the second test, let us now consider a completely random 100 gate circuit and then do the same process. Note that we no longer are able to complete randomize the circuit as gates will now collide.

For the third and final test, we can consider a 500 gate random circuit. The results of all the tests are below.

Scenario	Method	Average
Non-colliding	Random ordering	0.4532510031
	Walking	0.4522365294
	Shooting	0.4535883884
100 gate	Walking	0.4286999006
	Shooting	0.4226936795
500 gate	Walking	0.2003583436
	Shooting	0.1988034497

Table 2: Average mean hamming distance by scenario and method

As we can see from the table, we see only very small differences between random shooting and random walking. Thus, we conclude that there is little gain in using random walking over random shooting given the additional algorithmic complexity in random walking.

For an idea of how well we can actually mix, below are two heatmaps. The first is a random 100 gate circuit on itself, which we expect to have a "red beam" along $y = x$ as the circuits are equal. The second is the same 100 gate circuit but with random shooting. We note that the heatmap for random walking looks extremely similar.

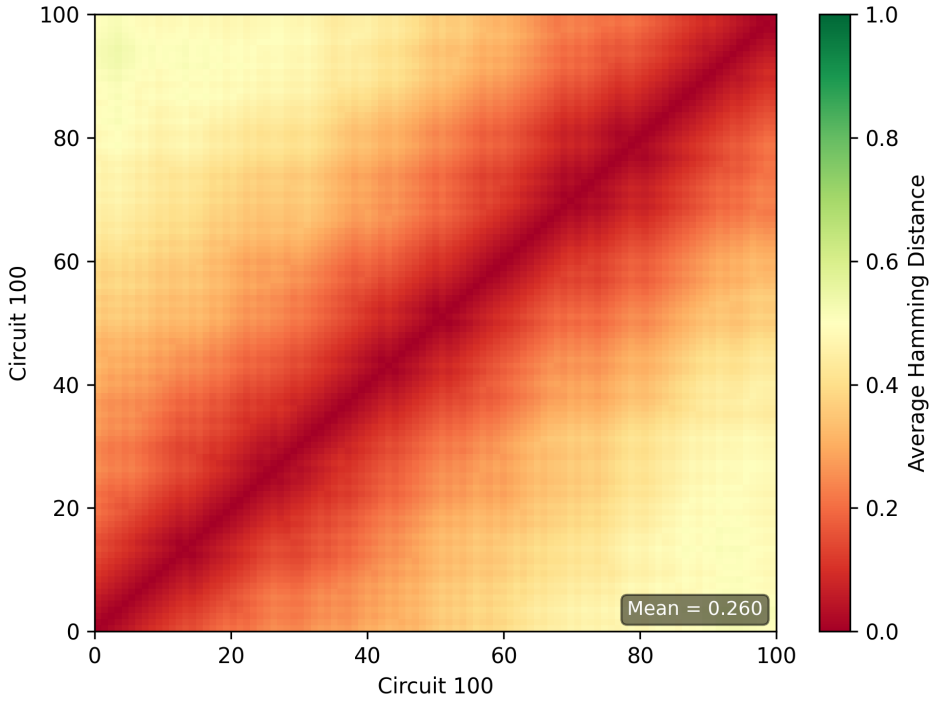


Figure 17: Heatmap showing a 100 gate circuit with itself

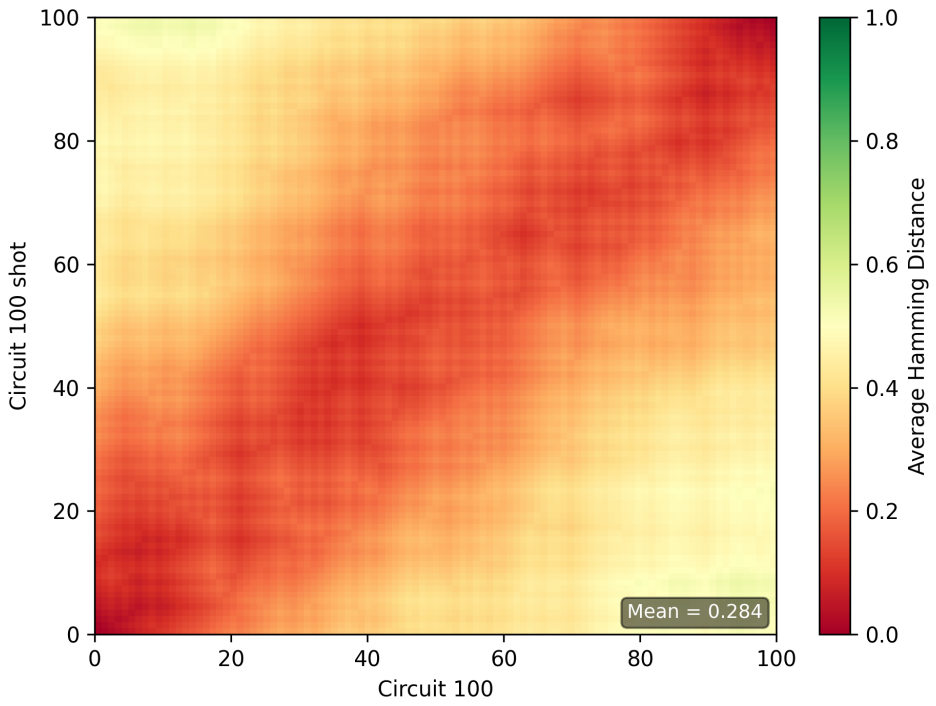


Figure 18: Heatmap showing a 100 gate circuit before and after random shooting

6.2 Replacing Gates with Rewired Identities

A more complex method of achieving blurring is to make single and pair gate replacements. The process for both is given below.

Single Gate Replacement

1. Select a random gate g.
2. Create a random identity I by finding two friends from our rainbow table.
3. Rewire the very first gate of I to match g to get I'. We note that this is still an identity as bit shuffles of identities are still identities.
4. Remove the first gate of I', which should be equal to g, to get B.
5. Replace g with B.
6. Repeat.

For the pair replacements, we will classify each type of overlap from the right gate onto the left gate. For instance, $123;321$ we can say that the active pin of the left gate's active pin shares a wire with the second control pin of the right gate, the left gate's first control pin shares a wire with the right gate's first control pin, and the left gate's second control pin shares a wire with the right gate's active pin. For pairs of gates, there are 33 different ways to have an overlap (excluding the case where the two gates don't overlap at all). This is because there are 6 ways to have an overlap on every single wire of the left gate, 18 ways to have 2 overlaps on the left gate, and 9 ways to have an overlap on a single wire. We can now define how to make pair replacements.

Pair Gate Replacements

1. Pair up all gates in the circuit and store their overlap types in a table T.
2. Create a random identity I by finding two friends from our rainbow table.
3. If I's taxonomy matches one in T, then rewire the first and second gate to match the pair to get I'.
4. Remove the first two gates of I' that match the pair with the matching taxonomy to get B'.
5. Replace the pair with B'.
6. Repeat until enough pairs have been replaced.

The biggest advantage to using the single gate replacements is the simplicity. It allows us to hide a single gate amongst 4 to 8 gates. However, the state before these 4 to 8 and after them will remain the same. Thus, by replacing a pair of gates, we get to remove the seam between those two gates. So while it is harder to replace a pair, as the best we can do is check a random identity and see if it matches, it is more effective in blurring. Thus, the first question is whether we can get enough randomness with only single gate replacements, similar to how we chose to use random shooting over random walking.

Below are two heatmaps of tests that use random shooting, ancilla replacements, single gate replacements, and some number of rounds of asymmetric butterfly.

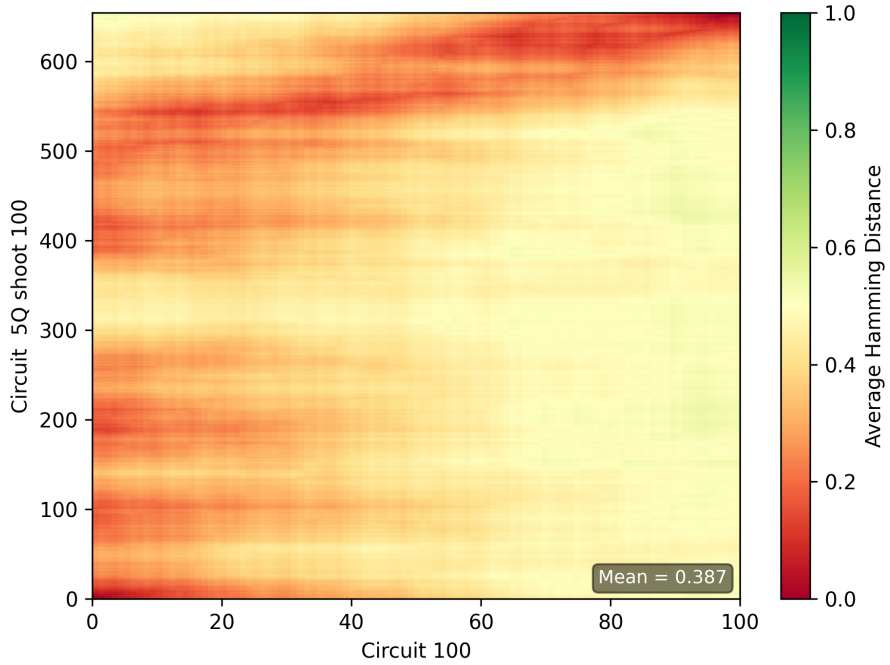


Figure 19: Heatmap showing a 100 gate circuit after random shooting, ancilla replacements, single gate replacements, combined in 5 rounds of asymmetric butterfly

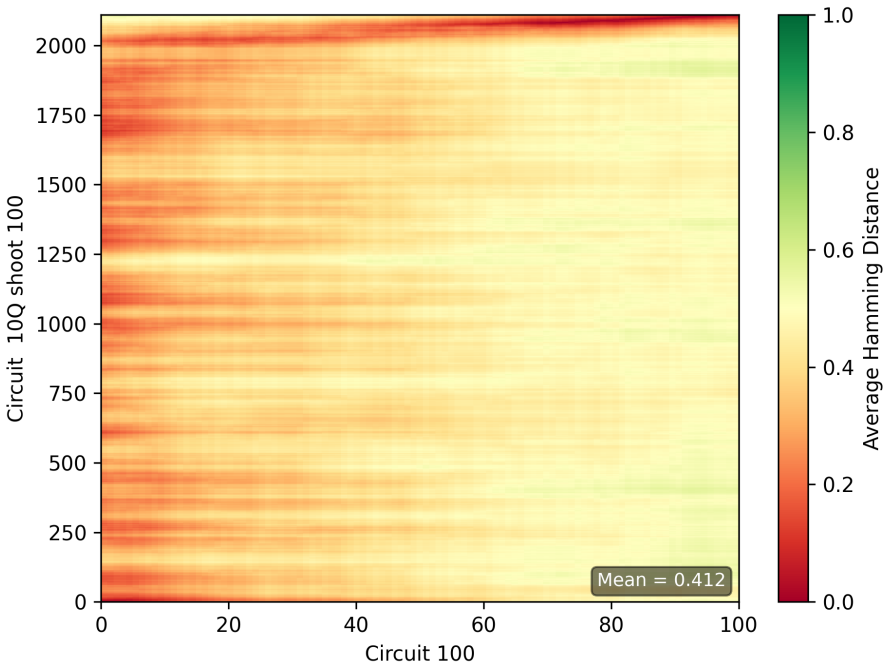


Figure 20: Heatmap showing a 100 gate circuit after random shooting, ancilla replacements, single gate replacements, combined in 30 rounds of asymmetric butterfly

While we get incompressibility from the above results, we see that there remains a "red beam" in the top right corner. In addition, for much of the obfuscated circuit, we see that it doesn't move much, indicated by the red lines on the left until near the top of the heatmap. We call this a red boomerang. We suspected that this was a result of the structure of the random 100 gate circuit, but upon testing with other random circuits, the red boomerang remained. Thus, it would appear that we need to test with pair gate replacements. The heatmap below shows the same test, but with only pair gate replacements.

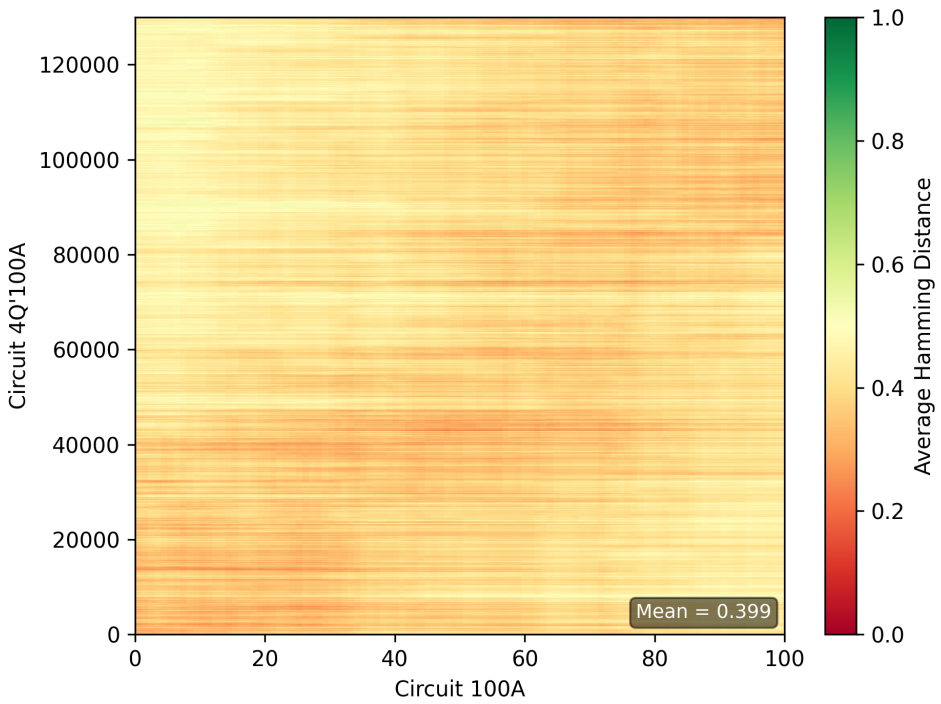


Figure 21: Heatmap showing a 100 gate circuit after random shooting, ancilla replacements, pair gate replacements, combined in 4 rounds of asymmetric butterfly

We quickly note that we have tested this a number of times and got similar results each time. We see no "red beam" anymore and the randomness is much more spread out. There is still a very spread out red "line" on $y = x$ and so additional blurring is needed. This is the only way we have found so far that has managed to get the blurring and incompressibility that we desire. We also believe that it would be good to use both single and pair gate replacements, rather than just one.

7 Methods on Pair Replacements

Above, we discussed a partitioned pair gate replacement. However, this doesn't actually provide much mixing outside of the replacements themselves. The gates adjacent to each other don't interact in any meaningful way, unlike how adjacent identities in the butterfly will attempt to randomize via shooting and compression. Thus, the methods that we discuss here will attempt to further introduce randomization amongst gates through the use of pair gate replacements. We note that all of these methods incorporate shooting somewhere in the algorithm as well. The point of these methods is to get away from the large blowup incurred from the butterfly methods, while still effectively mixing our circuits.

There are a number of methods that we have tried with motivation of pair get replacements. We note that pair gate replacements are actually very similar to the ancilla replacements that we discussed before, as a pair could span anywhere from 3 to 6 wires. Thus, all we need to do is make a replacement on 6 or 7 wires in order to use ancilla wires.

7.1 Replace Pairs Sequentially

In this method, we make pair replacements in a sequential manner before doing compression. For short, this method is called RCS. RCS is very similar to the method described above. However, we no longer partition our circuit into pairs. Instead, we do the following

1. We take $g_1 g_2$ as a pair and replace it with $u_1 u_2 \dots u_l$
2. We then treat $u_l g_3$ as our next pair and replace it with $w_1 w_2 \dots w_k$
3. Our next pair is $w_k g_4$ and do the same
4. Repeat until we have reached the end of the circuit.

Notice that our end result is a sequence of gates that never return to the original functionality of $g_i g_{i+1}$ because we replace it with $u_1 u_2 \dots u_l$, but then use u_l in a later pair swap. Notice that it is possible that g_1 and g_2 do not collide at all. In fact, they may not even share any of the same wires. We experimented with some ideas on forcing collisions and while we didn't see any new results, we will state the ideas below for completion.

1. We take $g_i g_{i+1}$ as a pair
2. If g_i and g_{i+1} share any wires, then replace with $u_1 u_2 \dots u_l$ as before. The next pair to consider is $u_l g_{i+2}$
3. Else, we shoot g_{i+1} to the right
4. If g_{i+1} eventually collides with some gate h_j, then we treat $h_j g_{i+1}$ as a pair and replace it with $w_1 w_2 \dots w_k$. The next pair to consider is $g_i g_{i+2}$
5. If g_{i+1} does not collide with any gate to the left, then make it a single gate replacement. The next pair to consider is $g_i g_{i+2}$

Notice that in the experiment above, we don't fully remove the $u_1 u_2 \dots u_l$ in the cases that we must shoot left. It remains unclear whether this is a huge loss or not and we did not explore this much further as little to nothing was lost when switching to the simpler method discussed at first. Below is a result of RCS after 3 rounds.

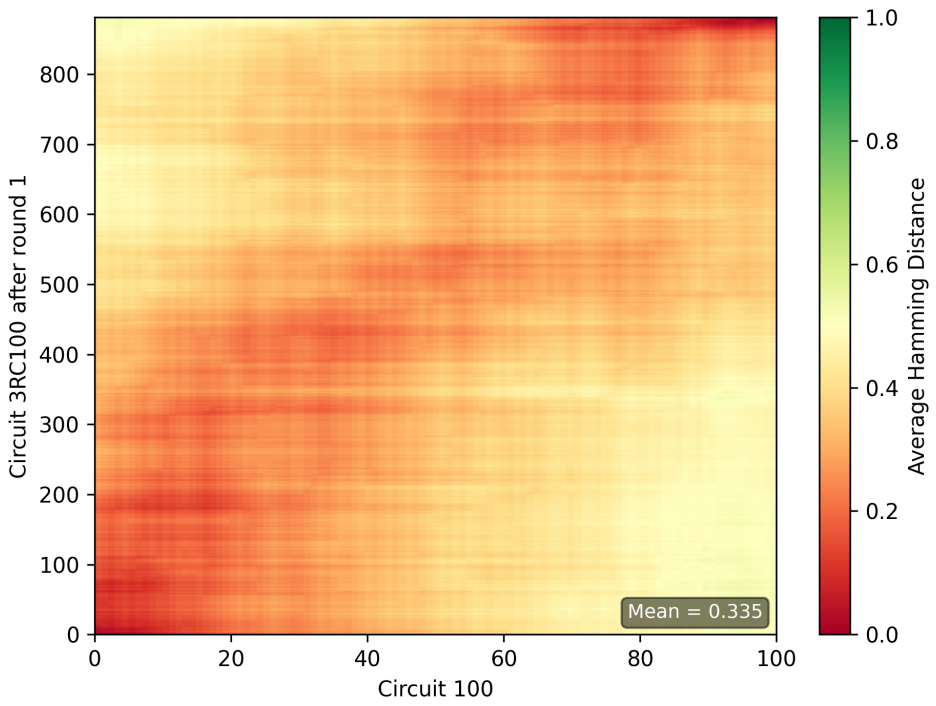


Figure 22: Heatmap showing a 100 gate circuit after 3 rounds of RCS

This heatmap is quite similar to the heatmap taken from our partitioned version of pair replacements. Of course there is more correlation overall, but the gate blowup is much smaller. In fact, we can take this to 50 rounds and still be only at a fraction of the number of gates that the pair gate replacements with asymmetric butterfly.

For comparisons sake to the next method, we will also show a heatmap of a test done on 16 wires.

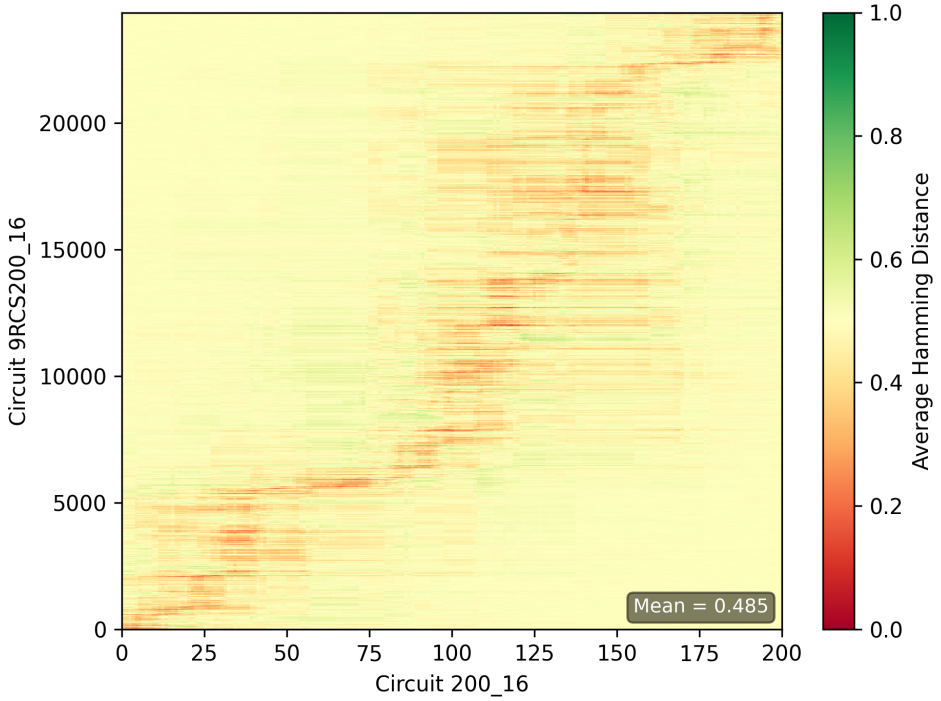


Figure 23: Heatmap showing a 100 gate circuit after 9 rounds of RCS on 16 wires

We will return to this method after discussing the other two methods.

7.2 Replace Pairs by Distance

One problem with the above method is that we replace pairs in a very standard manner, without caring for the kinds of pairs that we replace. For instance, suppose a circuit is extremely randomized in the first half and completely untouched in the second half. Then ideally, we would further randomize the second half in order to get complete randomness. The sequential method constantly gets close to the original states, and we wish to stay away from these points until the end of the circuit. This is the idea of the replace pairs by distance method (RCD).

This method records of notion of "distance" in between each gate. These distances measure how far we have gone from the original states of our original circuit. Suppose we have a circuit $g_1g_2g_3$, then we can record 4 states in total. Before g_1, between g_1 and g_2, between g_2 and g_3, and after g_3. We start with all of these states being 0. We then make our initial sequential pass in order to "erase" the 0s. So replace g_1 and g_2 as $u_1u_2u_3u_4$. We then end up with the circuit $u_1u_2u_3u_4g_3$. Notice that after u_4 the circuit returns to the state imposed by g_1g_2, and so the state between u_4 and g_3 remains 0. However, the states in between u_iu_{i+1} are all different. u_1 is 1 gate away from the original state, as is u_4, while u_2 and u_3 are two gates away. This yields our list of states to be $[0, 1, 2, 1, 0, 0]$. After replacing u_4g_3 with $w_1w_2 \dots w_8$, our states become $[0, 1, 2, 1, 1, 2, 3, 4, 3, 2, 1, 0]$. Now notice on the left hand side we have an ascending $[0, 1, 2]$ as a result of have $u_1u_2u_3$ as our first three gates. If we were to replace u_1u_2, this ascending nature will not change, as we can never be more than 1 gate away from the original state for our first gate. In other words, our gates at the beginning are already as far from the original state as they can be. Thus, we can ignore these. The same can be said for the descending nature at the very end. Thus, the only states we should consider for future replacements are $[1, 1, 2, 3]$. Notice that we have "erased" the 0s in this section. We can continue and remove

the 1s, then the 2s, etc. Ideally, we would be able to continue until we have removed all 30s, but practically this becomes very hard due to gate blowup. Our tests have thus been after removing all 10s.

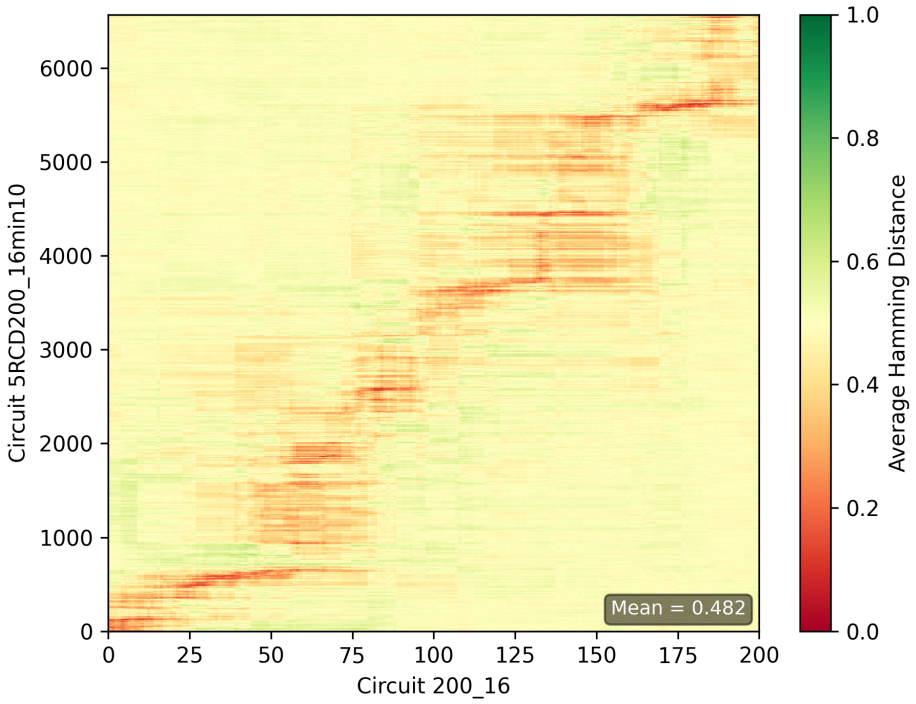


Figure 24: Heatmap showing a 100 gate circuit after 5 rounds of RCD on 16 wires

The above heatmap looks extremely similar to that of the RCS method. With further rounds, we only get a lighter red line, but can't seem to escape the red line that reveals our correlation. In addition, we clearly notice that the line is much clearer in the 16 wire version than the 64 wire version. This reveals that our randomization methods are merely fixing the blurring that we have from our shooting. On 16 wires, shooting is much less effective as gates will collide much more often. Thus, it is clear we need some further form of randomization. That is not to say that the above methods are useless though, as we have still achieved incompressibility. At this point, we are most concerned with the heatmap attacks.

7.3 More on RCS and RCD

One thing that we have retained throughout all of our experiments, is maintaining functionality of the circuit. As a result, we have always gotten a strong diagonal on our heatmaps due to the functionality being equivalent, causing there to be guaranteed strong correlation in the corners along $y = x$. To better understand our results, it would first be ideal to understand how many gates we need in order for completely random circuits to look random in a heatmap. Below, we compare two circuits with varying wires and gates in hope of understanding how heatmaps change as we increase the number of wires and increase the number of gates.

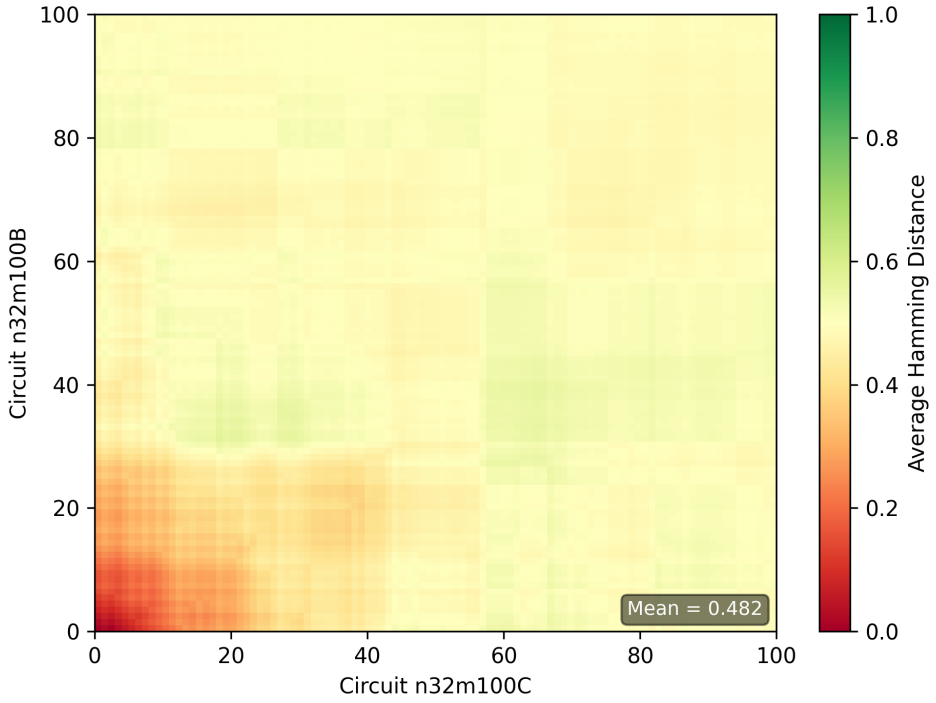


Figure 25: Heatmap two completely random 100 gate circuits on 32 wires

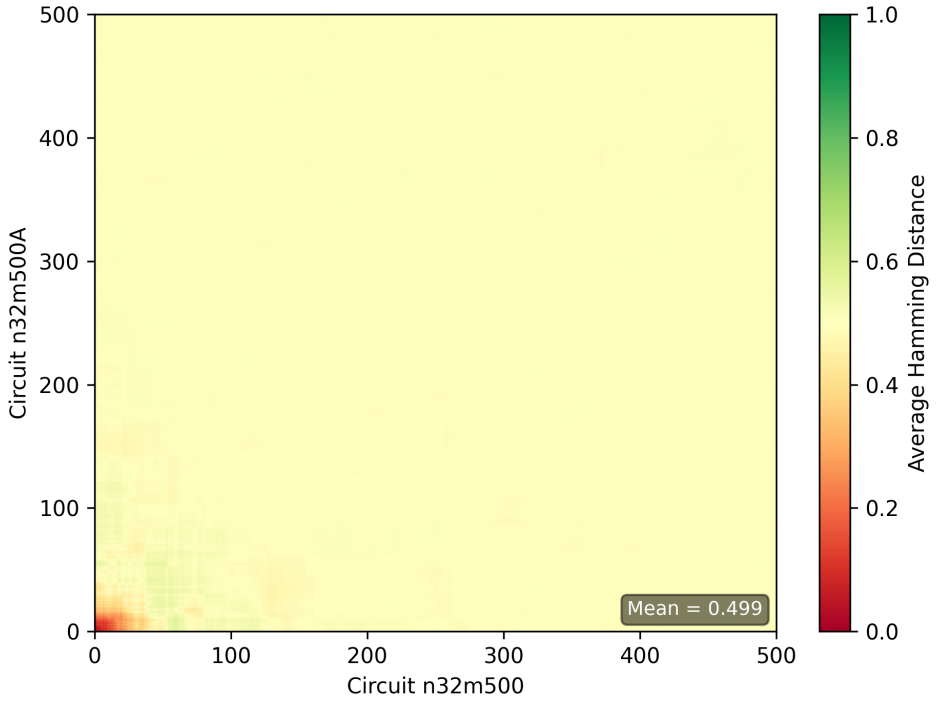


Figure 26: Heatmap two completely random 500 gate circuits on 32 wires

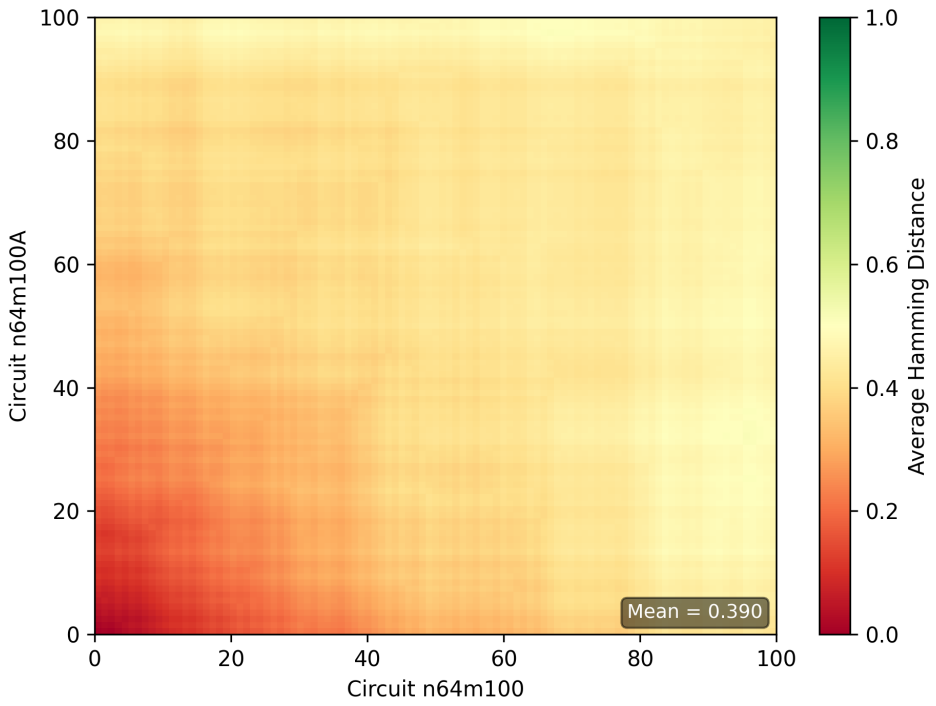


Figure 27: Heatmap two completely random 100 gate circuits on 64 wires

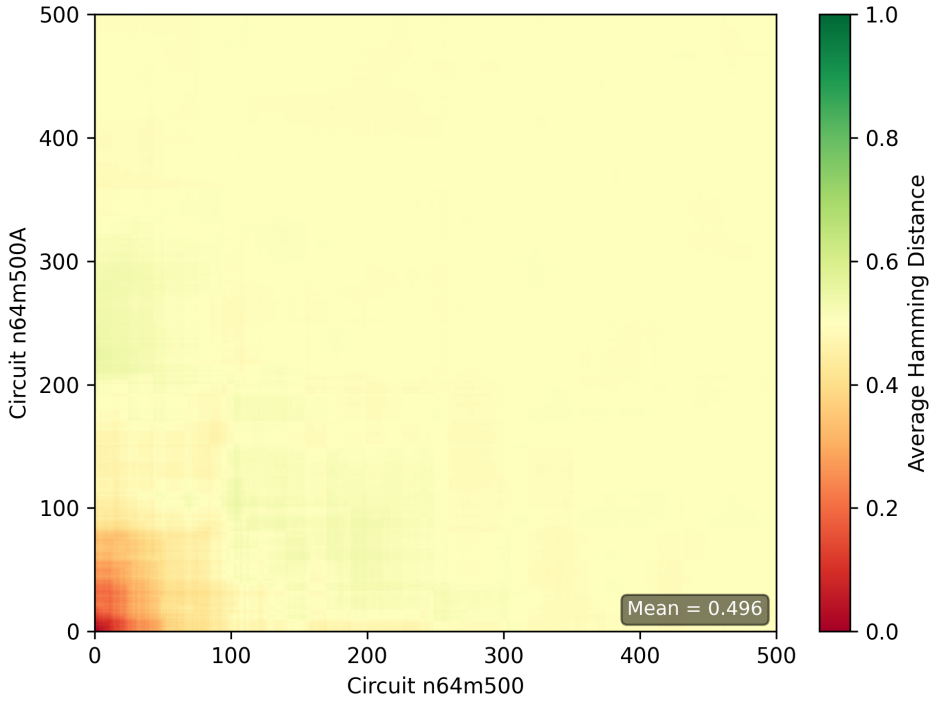


Figure 28: Heatmap two completely random 500 gate circuits on 64 wires

From the above, we can see that on 32 wires, the heatmap for two completely random circuits already looks uncorrelated with 100 gates. On the other hand, we have a good bit of correlation on the 64 wire case with 100 gates. The randomness returns when we look at the case with 500 gates. Let us now take an example of RCS/RCD on 100 gates and 128 wires.

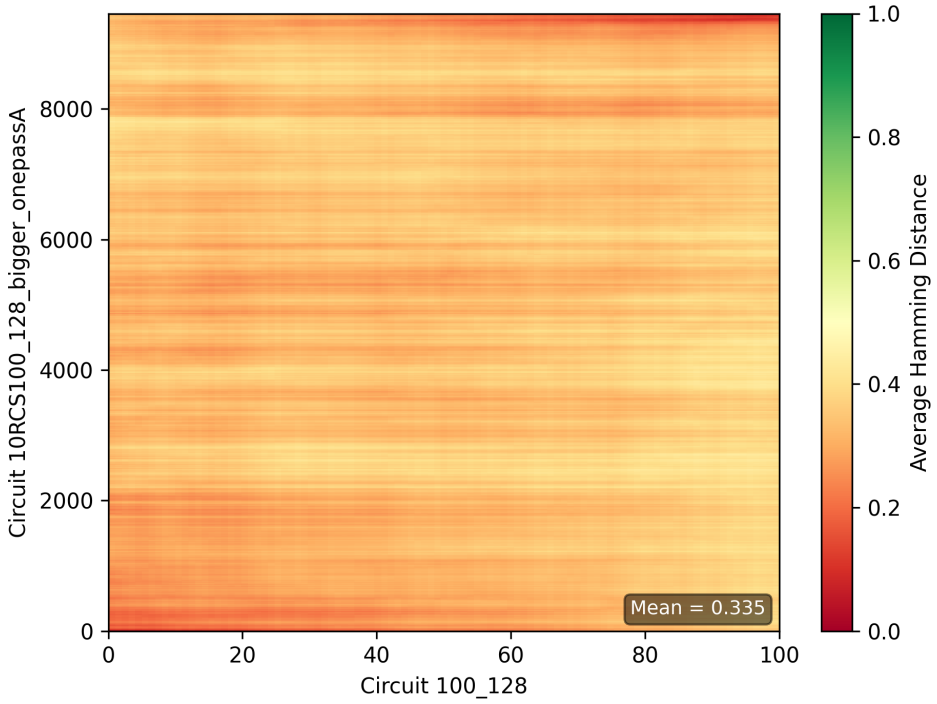


Figure 29: Heatmap showing a 100 gate circuit after 10 rounds of RCS on 128 wires

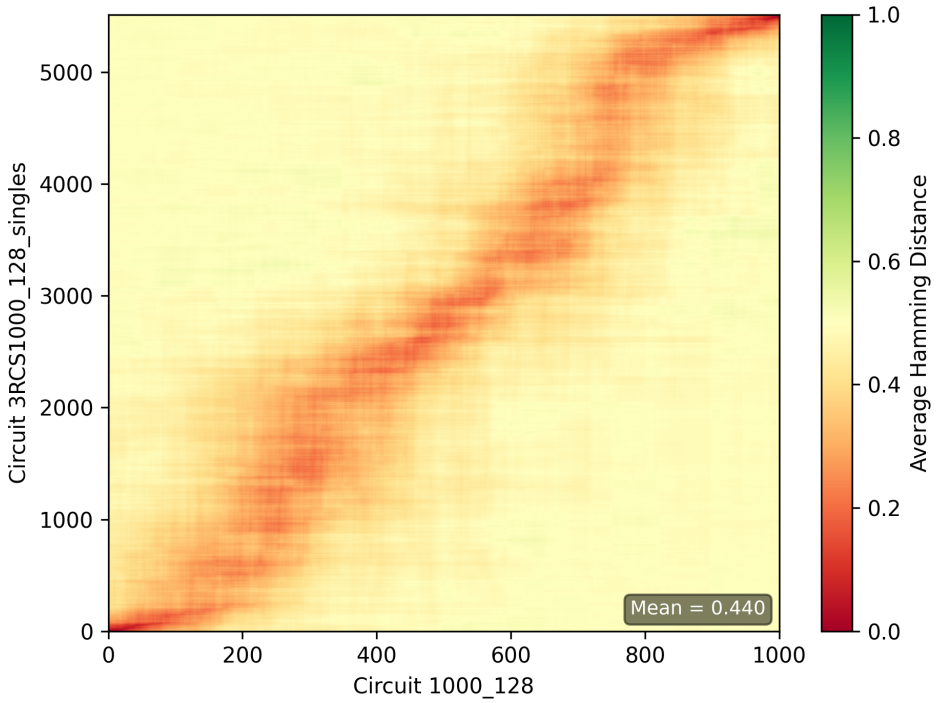


Figure 30: Heatmap showing a 1000 gate circuit after 3 rounds of RCS on 128 wires

From the above, we can see that while the 100 gate example has quite a blurry heatmap, the 1000 gate one reveals that the correlation remains extremely clear. Thus, our results on 100 gates, at least on 100 wires, is insufficient. One idea that comes from this is given our starting number of gates, to simply increase the number of wires that we work on in hopes that we can force a heatmap that blurs in the way we saw previously. However, this would continue to limit us, as even those blurred heatmaps showed a lot of correlation. This correlation is, in a way, expected as for equivalent circuits, we will always have correlation in the corners along $y = x$. Thus, one question to ask is if we really need to maintain equivalence?

7.4 Pre-Interleaving

In order to play around with this idea, we must first establish that we can not just destroy all semblance of equivalence, as in the end, an obfuscated circuit should still be able to used for its original purpose. Despite this goal, we can still create a random circuit via pre-interleaving.

The idea of pre-interleaving is to create a circuit that is completely random overall on $2n$ wires, but on the first n wires maintains functionality. A simple way to do this is with the following:

1. Let C be a random circuit on n wires with m gates
2. Generate a random C' on n wires and m gates and then shift to wires to lie on $n..2n$ as opposed to $0..n$
3. Interleave C and C' together to get a new circuit D

From this, we have that C' gives us a completely random circuit D, but as we shifted all of its gates to commute with the gates of C, then the functionality of D on the first 64 wires will depend solely on C. In other words, given D, we can just limit our view to the first 64 wires in order to use it as intended. The

hope here is to not hide the fact that we are using only half the wires, nor is it to hope that we can hide which 64 wires we have chosen. Instead, we can combine this interleaving idea with our previous RCS/RCD methods. As we have interleaved our gates together, then each adjacent pair will be noncolliding. When we replace them with our identities, we hope to enforce some collisions and, in a way, fix these gates together such that the interleaving can not be undone. This is very similar to our hope of fixing our blurring methods from earlier. In addition, by allowing a completely random C' to help create a D, a heatmap of D and D' would, in theory, look completely random.

Unfortunately, we did not see great results from this method. The heatmap below reveals that we can still easily correlate between the original circuit and the new circuit. We tested a number of heatmaps, and they all revealed the underlying correlation. We tried first to take the inputs only on the first 64 wires. If things worked as we thought they would, then the additional C' would incur great randomness on these 64 inputs. After, we took a heatmap on 128 wires, which again, should be quite difficult to relate to the original circuit due to the circuits not even being correlated.

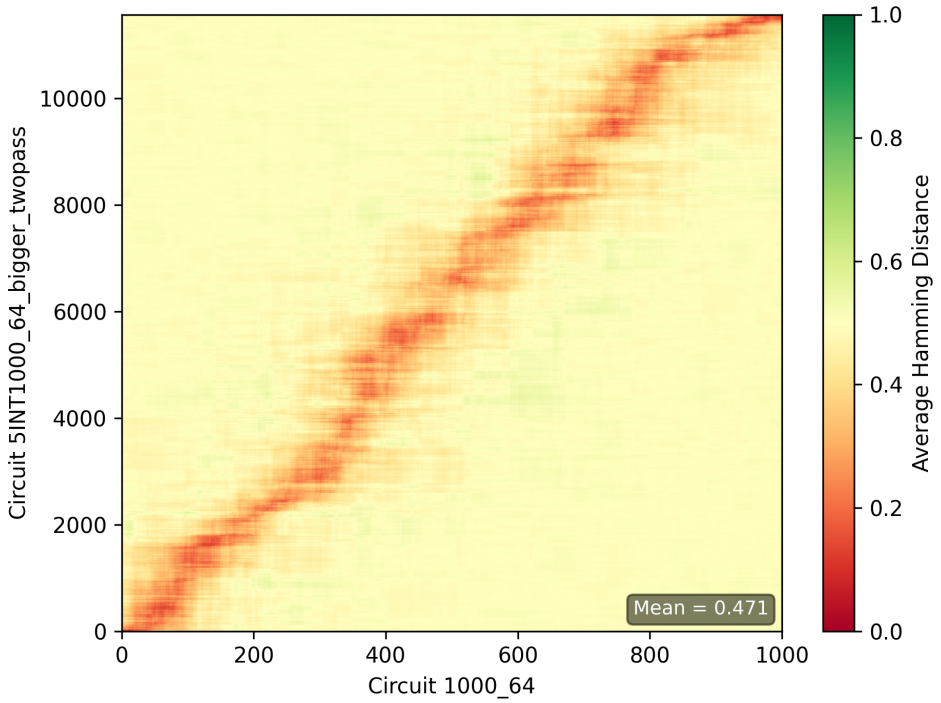


Figure 31: Heatmap showing a 1000 gate circuit, interleaved with a second 1000 gate circuit, using inputs on 64 bits

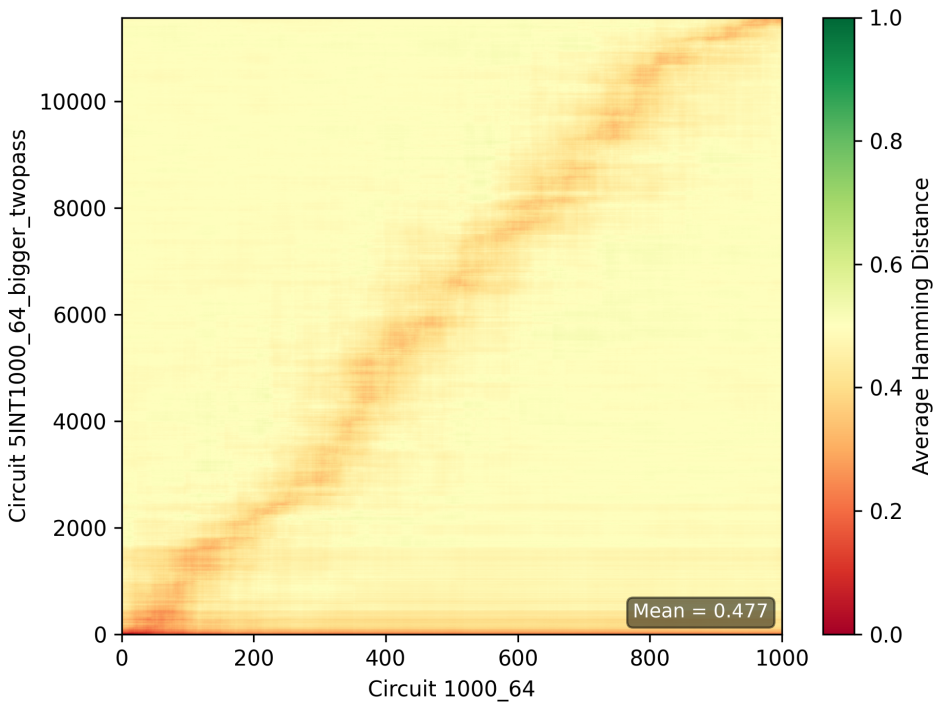


Figure 32: Heatmap showing a 1000 gate circuit, interleaved with a second 1000 gate circuit, using inputs on 128 bits

In a way, this might not be too unexpected. The butterfly methods that we worked on before, were initially tested on identities over 5 wires. Our results in the end, would constantly look correlated no matter how much we increased the wires, but that was largely due to our resulting circuit being heavily condensed on the original 5 wires. From this, it seems that adding gates on new wires is ineffective in randomizing a circuit. So while we leave the interleaving method largely untested from these results, we still find promise in this idea of "leaving functionality behind".

8 Wire Shuffles and Bit Flips

We take a different approach here. We recall that we have already achieved the goal of incompressibility and only need to beat the heatmap attacks (at least for now). The heatmaps simply record hamming distances between states as they evolve throughout the two circuits. Thus, one simple way to force this to differ greatly, is to shuffle the wires that we are working on. If we are to do this, then we have artificially beaten the heatmap attacks, as two different wire shuffles on the same circuit C, would result in C having a higher hamming distance than if we took a heatmap of C with itself.

Let us think about how we can achieve these shuffles. One way that utilizes a similar thought to our butterfly method, is to do the following.

Butterfly-like

1. Start with some circuit $C = g_1 g_2 g_3 \dots g_m$
2. For g_i, replace it with some shuffle $B_{w_i} g_i' B_{w_i}^*$ such that funtionality is maintained.

The above would yield something like $B_{w_1} g_1' B_{w_1}^* B_{w_2} g_2' B_{w_2}^* \dots$. One of the advantages of using our wire

shuffles, is that they form a subgroup of all circuits. In addition, it is much easier to find equivalent wire shuffles than it is to find equivalent circuits. Thus, in mixing our shuffles, we can actually find a random, yet equivalent, circuit representation of $B^*_{w_1} B_{w_2}$. Thus, we would ideally never return to the original states until we reach the very end of the circuit. From what we see below, heatmaps reveal that we can not actually get far enough from the original functionality for this to be the case. The many red horizontal lines show that we are constantly returning back to the original functionality, or at least getting close enough to it to reveal correlation.

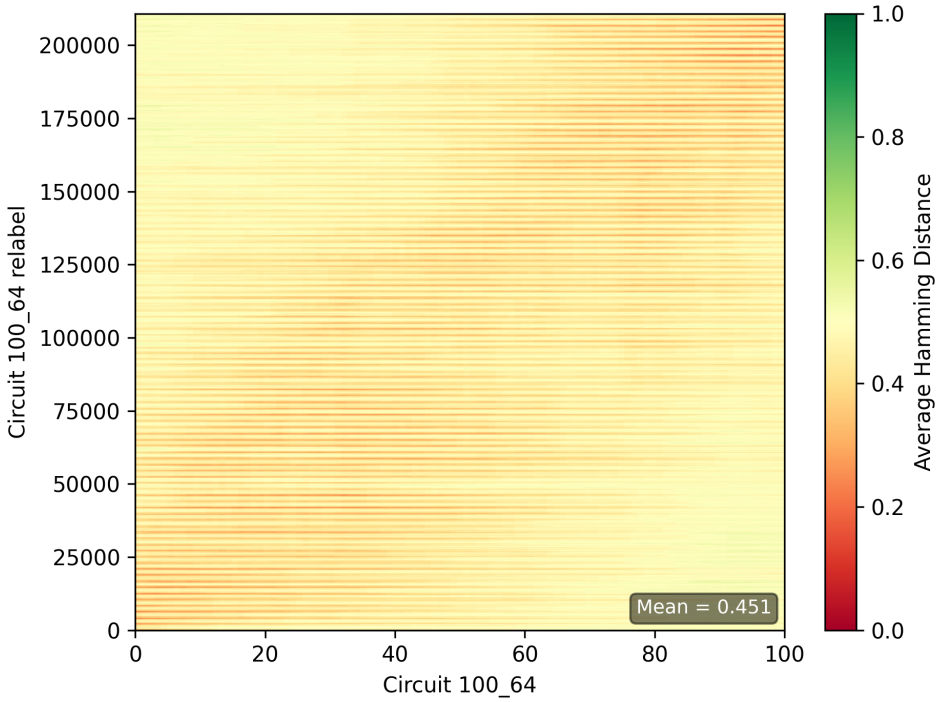


Figure 33: Heatmap showing a 100 gate circuit on 64 wires after a butterfly-like shuffle of wires

Thus, we need to carry our wire shuffles throughout the entirety of the circuit. Let us take this in the simplest way possible.

Singular Shuffle

1. Start with a circuit $C = g_1 g_2 g_3 \dots g_m$
2. Insert a shuffle at the very beginning and at the very end, shifting g_i to g'_i such that $g_1 g_2 g_3 \dots g_m = B_w g'_1 g'_2 g'_3 \dots g'_m B^*_w$

Of course, as before, we can get two random circuits that compute B_w so that we are not merely reversing the same circuit. Thankfully, our sanity checks pass and our heatmap now looks extremely random.

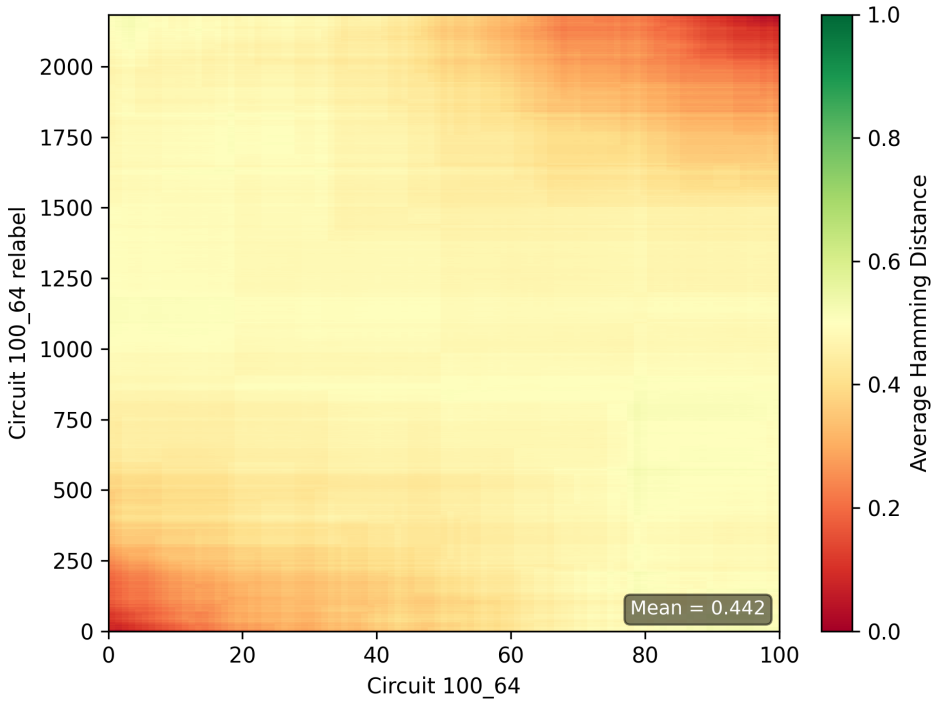


Figure 34: Heatmap showing a 100 gate circuit on 64 wires after a simple shuffle of wires

The main reason that this works so much better, is that in no point in the circuit, are we returning to the original functionality. We remain far from it with our first shuffle and do not return until the very end. Of course, this does not hide anything. An attacker can easily look at our circuit and then see that our initial gates are merely a shuffle amongst the wires, undo the shuffle, and then recover the original circuit. Thus, we need to do something more than just a simple singular shuffle. Let us do the following instead.

Gate-by-gate Shuffles

1. Start with a circuit $C = g_1 g_2 g_3 \dots g_m$
2. For each gate, insert a B_{w_i} and shift g_i as needed to maintain functionality to get

$$B_{w_1} g_1' B_{w_2} g_2'' B_{w_3} g_3''' \dots B_{w_m} g_m^{(m)'} B^*,$$

   where B^* is the inverse of the composition of all B_{w_i}.

This alone makes it much harder to find our original gates g_i. If we combine this now with our previous mixing methods, such as RCS, then we can ensure that this is also incompressible. This also gives us some randomization amongst the swap gates to hide the fact that they are doing swaps or even hiding the original gates into these swaps, making it even harder for an attacker to determine which gates are part of swaps and which gates are from the original.

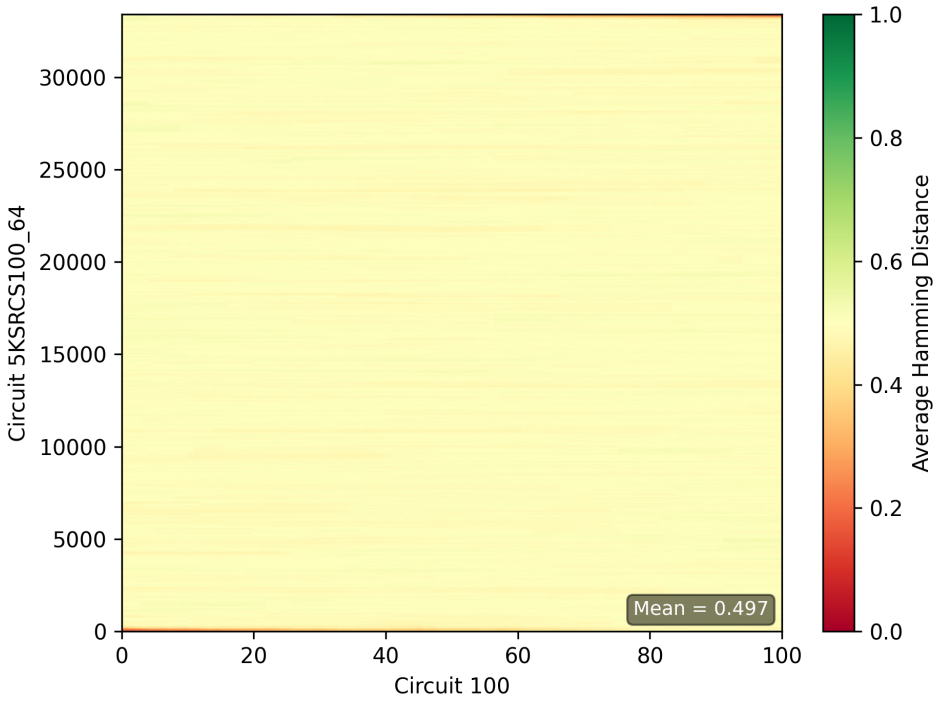


Figure 35: Heatmap showing a 100 gate circuit on 64 wires after gate-by-gate shuffles

As seen above, the heatmap remains close to the ideal that we are looking for. One thing to note is that the blowup in the number of gates is much greater. As we are limited to gate r57, in order to compute a swap of two wires, it requires at minimum, 6 gates and 3 wires. The algorithm above utilizes a random choice of a swap using 3 wires and between 6 and 20 gates, just for some additional randomness. In order to compute a completely random shuffle on the wires, we can then use a Knuth shuffle. This requires n swaps in total, which gives us a total of $6n$ blowup in the total number of gates. Thus, one thing that one could try is to instead only add wire shuffles to some gates, rather than before every single gate. Then repeat it in combination with RCS and compression. In other words,

1. Start with a circuit $C = g_1 g_2 g_3 \dots g_m$
2. Insert x number of shuffles in between the gates C, as well as before g_1 and after g_m.
3. Mutate the shuffled C via RCS
4. Compress
5. Repeat

The shuffle before g_1 is meant to ensure that the entire circuit is shuffled, at least a little. The shuffle after g_m is meant to undo everything and maintain functionality. One advantage to this relies on the fact that adding shuffles without RCS remains compressible. For instance, a gate-to-gate shuffle insertion can blow up a 100 gate circuit to a 60,000 gate circuit, on 64 wires. This compresses down to around 31,000 gates. Thus, we believe that we can add many shuffles and compress them along the way, such that we have both many shuffles as well as minimal gate blowup. However, this remains untested.

These heatmaps are not perfect. Taking a closer look, we can still see a number of, though faint, red horizontal lines. As we are simply using shuffles, we do not actually make any significant changes to the

circuit itself. One way that we can introduce some functional changes to the circuit, is by introducing bit flips. We note that the set of all circuits which swap as well as flip bits, remains a subgroup of all circuits. Thus, we can do all of our methods above, but now allow our swaps to also flip a single bit. More testing is needed to understand how much we gain from this, but an initial glance tells us that we are indeed getting more randomness in our circuits.

9 Conclusion

We conclude this with our current best results. Namely, that is to use repeated, and limited, gate-by-gate wire shuffles, RCS, shooting, as well as compression. Ideally, we would be able to conduct hundreds or even millions of rounds of this, however, we are unable to do so in a practical manner. Due to the inefficiency of constructing swaps with only gate r57, our blowup on n wires remains $\times 6n$, giving us a quadratic blowup. In addition, as more rounds are used and our cicuits continue to grow, compression becomes harder and harder, requiring more time and memory. Our largest bottleneck when it comes to speed is sampling random convex subcircuits.

We finally discuss some open problems. It would also be important to expand our rainbow tables further. After all, our random replacements are currently limited by our rainbow tables. As seen in the gate replacement methods, it is important that we sample random identities past stupid identities. It is unclear whether we need the identities to be incompressible. We have ideas for incompressible identities that have a high amount of structure to them. This is done by creating a small staircase of gates and then repeating them until an identity is achieved. This can be done by calculating the order of the permutation. This highly structured periodic nature makes it extremely easy to identify and hard to randomly shoot.

Shuffling wires and flipping bits is only one way to step away from the original functionality and then returning. There may be other, much more interesting ways, to achieve this that we are still looking for. One concern with our method is the high structure may remain for attackers to exploit.

Finally, in order to analyze our results, we primarily look towards heatmaps and incompressibility. It would be of greatest importance to define more definitive security metrics for our algorithms to better understand the effectiveness of our results.

References

[1] R. Canetti, C. Chamon, E. Muccilio, A. Ruckenstein, *Towards general-purpose program-obfuscation via local mixing*, 2024.
`https://eprint.iacr.org/2024/006`