Cycle Slicer: An Algorithm for Building Permutations on Special Domains

Sarah Miracle and Scott Yilek

University of St. Thomas, St. Paul, USA
{sarah.miracle, syilek}@stthomas.edu

Abstract. We introduce an algorithm called Cycle Slicer that gives new solutions to two important problems in format-preserving encryption: domain targeting and domain completion. In domain targeting, where we wish to use a cipher on domain $\mathcal{X}$ to construct a cipher on a smaller domain $\mathcal{S} \subseteq \mathcal{X}$, using Cycle Slicer leads to a significantly more efficient solution than Miracle and Yilek's Reverse Cycle Walking (ASIACRYPT 2016) in the common setting where the size of $\mathcal{S}$ is large relative to the size of $\mathcal{X}$. In domain completion, a problem recently studied by Grubbs, Ristenpart, and Yarom (EUROCRYPT 2017) in which we wish to construct a cipher on domain $\mathcal{X}$ while staying consistent with existing mappings in a lazily-sampled table, Cycle Slicer provides an alternative construction with better worst-case running time than the Zig-Zag construction of Grubbs et al. Our analysis of Cycle Slicer uses a refinement of the Markov chain techniques for analyzing matching exchange processes, which were originally developed by Czumaj and Kutylowski (Rand. Struct. & Alg. 2000).

Keywords: Format-preserving encryption, Small-domain block ciphers, Markov chains, Matchings

1 Introduction

Block Cipher designers have traditionally been concerned with constructing ciphers that encrypt bitstrings of a particular, fixed length. The canonical example of such a cipher is AES, which encrypts 128 bit plaintexts into 128 bit ciphertexts. Recently, the practical interest in *format-preserving encryption* (FPE) schemes [2, 6, 23] for use on data like credit card numbers and US Social Security Numbers (SSNs) has led to the need for ciphers that do not just work on bitstrings of some length, but have more customized domains.

An FPE scheme should have ciphertexts with the exact same format as plaintexts. For example, social security numbers are 9 decimal digit numbers with many additional restrictions, so an FPE scheme for SSNs should result in ciphertexts that are also valid SSNs. Since any 9 digit number can be represented with 30 bits, using a block cipher with block size 30 bits[1] would be an obvious

[1] Even constructing a block cipher with such a small block size and with strong security guarantees is an interesting problem. See [4, 12, 19, 21] for more details.

first approach. However, encrypting 30 bits that represent a valid SSN could result in a 30 bit ciphertext that does not correspond to a valid SSN, or even a 9 digit number at all!

FPE is important in practice since practitioners often want to introduce encryption into an existing system while working within constraints like avoiding conflicts with legacy software or existing databases. The first FPE scheme is due to Brightwell and Smith [6], while the term format-preserving encryption was later coined by Spies [23]. Practical interest in this problem has led to a number of recent papers on FPE [2,4,11,15–17] and related papers on constructing new, provably-secure small-domain ciphers [12, 18, 19, 21] that can be used for FPE. The National Institute of Standards and Technology (NIST) has also published a standard for FPE [9] with schemes FF1 [3] and FF3 [5] based on Feistel networks, though recent papers [1,8] have introduced attacks on these standardized schemes.[2]

While many of the recent papers mentioned above focus on the practical issues surrounding FPE, they also raise a number of interesting, more theoretical problems. In this paper, we focus on the theory underlying two important problems in FPE, domain targeting and domain completion, and also describe how our results could be extended to a number of related problems.

Domain Targeting. Consider again the example above in which we would like an FPE scheme for valid social security numbers. One strategy would be to try to construct a specialized cipher that works on the exact domain we are interested in. The problem with this strategy is that this specialized cipher is specific to one domain and we would likely need to develop a new one if we later needed to encrypt valid credit card numbers, or valid email addresses of some length, or some other uniquely-formatted plaintexts.

If the desired domain has an efficient way to rank and unrank elements, then another solution is to use the *rank-encipher-unrank* approach of [2]. Specifically, to encipher a point in $\mathcal{S}$, this approach first applies a rank algorithm that maps elements in the target domain $\mathcal{S}$ to points in $\{0, \ldots, |\mathcal{S}| - 1\}$, then applies a cipher with domain $\{0, \ldots, |\mathcal{S}| - 1\}$, and finally applies an unrank algorithm to map the result back to $\mathcal{S}$. Efficient ranking and unranking algorithms are known for a number of domains that are important in practice [2, 10, 15, 16], for example, regular languages that can be described by DFAs. Nevertheless, there are domains without efficient rankings [2], and even domains specified by complex regular expressions can prove problematic with this approach. Additionally, the ranking and unranking algorithms cited above can also potentially leak timing information which, looking ahead, is something we would like to avoid with our results. Thus, it is useful to have alternative solutions that only assume the ability to efficiently test membership in $\mathcal{S}$ and do not rely on ranking.

A different strategy that does not rely on efficient ranking is to come up with a general method of transforming a cipher on a larger domain into a cipher on only a subset of the domain points. For example, in the case of social security numbers

[2] Bellare, Hoang, and Tessaro [1] give message recovery attacks on both FF1 and FF3, while Durak and Vaudenay [8] detail a more damaging attack on FF3.

such a method could transform a cipher on 30 bit strings into a cipher on 30 bit strings that encode valid SSNs. We call this problem of transforming a cipher on domain $\mathcal{X}$ into a cipher on domain $\mathcal{S} \subseteq \mathcal{X}$ *domain targeting*. This problem is well-known and good solutions exist. A common solution is to use *Cycle Walking*, a folklore technique first formally analyzed by Black and Rogaway [4]. With Cycle Walking, given a permutation P on $\mathcal{X}$, we can map a point $x \in \mathcal{S} \subseteq \mathcal{X}$ by repeatedly applying P until we hit another point in the target set $\mathcal{S}$. In other words, we first check if $P(x) \in \mathcal{S}$, then check $P(P(x)) \in \mathcal{S}$, and so on. Because permutations are made up of cycles, the procedure will eventually either find another point in $\mathcal{S}$ or traverse an entire cycle and return back to x.

Unlike the ranking solutions discussed above, Cycle Walking only requires the ability to test membership in the target domain $\mathcal{S}$. It can also be a good option in practice, since the expected time to encipher a point is small. For example, if the size of the target set $\mathcal{S}$ is at least half the size of the larger set $\mathcal{X}$, then the expected number of applications of the permutation P is at most 2. The worst-case running time of Cycle Walking, however, is much worse, since we might need to walk through a long cycle of points in $\mathcal{X} - \mathcal{S}$ before finding another point in $\mathcal{S}$. The fact that the running time also depends on the specific point being enciphered could also potentially cause headaches for practitioners, due to unpredictable running times or subtle leakage of timing information.

To address these issues, Miracle and Yilek recently introduced an alternative to Cycle Walking called *Reverse Cycle Walking* (RCW) [17]. RCW has lower worst-case running time than traditional Cycle Walking, and the running time does not vary depending on the input. The basic idea underlying RCW is to use permutations on $\mathcal{X}$ to form matchings on $\mathcal{X}$. If, when applying a permutation P on $\mathcal{X}$, two points x and x' from $\mathcal{S}$ appear consecutively in a cycle and the preceding and following points are not in $\mathcal{S}$, then x and x' are matched, and potentially swapped depending on a separate bit flip. Thus, in each round of RCW, many points from $\mathcal{S}$ swap positions. Miracle and Yilek apply a result of Czumaj and Kutylowski [7] to show that repeating this process for enough rounds leads to a random mixing of the points in $\mathcal{S}$.

One disadvantage of RCW is that the number of rounds needed quickly increases as the size of $\mathcal{S}$ gets closer to the size of $\mathcal{X}$. In the same social security number example discussed above, $|\mathcal{S}|/|\mathcal{X}| = 10^9/2^{30} \approx .93$, which is well above $1/2$, so RCW performs poorly, requiring millions of rounds to mix $\mathcal{S}$ sufficiently. The reason for the slowdown is that RCW only swaps points in $\mathcal{S}$ when they appear in a cycle sandwiched between points from outside of $\mathcal{S}$. Specifically, RCW will only potentially swap $x, x' \in \mathcal{S}$ if they are part of a cycle $(\ldots\, y\ x\ x'\ z \ldots)$ with $y, z \notin \mathcal{S}$. However, if the size of $\mathcal{S}$ is close to the size of $\mathcal{X}$, then we are likely to have cycles that instead have many consecutive points from $\mathcal{S}$ and few points from $\mathcal{X} - \mathcal{S}$, making swaps unlikely with RCW.

Looking forward, our main result will be a new algorithm we call Cycle Slicer that will allow us to take long cycles of points from $\mathcal{S}$ and form many matchings. This will allow us to substantially improve upon RCW for the important cases where $|\mathcal{S}|$ is close to the size of $|\mathcal{X}|$. In particular, in the SSN example, Cycle

Slicer will only need around 12,000 rounds, which is significantly less than needed by RCW.

Domain Completion. The second problem we study is *domain completion*, which has its roots in ad-hoc solutions practitioners used in place of FPE, and which was recently given a formal treatment by Grubbs, Ristenpart, and Yarom [11]. Before practical FPE schemes were available, practitioners would solve their ciphertext formatting issues using tokenization systems. Essentially, they would construct a permutation on their desired domain by populating a table with input-output pairs. For example, to encipher an SSN without a good FPE scheme, one can instead add the SSN to a table and randomly sample another social security number to map it to. In other words, in the absence of a good FPE scheme on their desired domain, practitioners would instead construct their own permutation on the domain by using what is typically known as lazy sampling.

If a system contains such a table of input-output pairs, but then at a later point in time a good FPE scheme becomes available for the desired domain, it makes sense to stop adding entries to the table and to start using the new scheme. Yet, to maintain backwards compatibility, the new FPE scheme should only be used on new points, and the table (which can now be made read-only) should still be used on old values. This practical problem leads to an interesting theoretical question: how can we construct a permutation on some domain while staying consistent with a table of existing mappings?

More formally, assuming it is easy to construct permutations on some domain $\mathcal{X}$, we wish to construct a new permutation on $\mathcal{X}$ that preserves an existing set of mappings for points in a *preservation set* $\mathcal{T} \subseteq \mathcal{X}$. Let $\mathcal{U}$ be the set of points that the table maps $\mathcal{T}$ to (i.e., the range of the table mappings). One of the challenges with domain completion is that $\mathcal{T}$ is unlikely to be the same as $\mathcal{U}$, yet the two sets might have some overlap. Thus, the problem is not as easy as just mapping points in $\mathcal{X} - \mathcal{T}$ to other points in $\mathcal{X} - \mathcal{T}$. Some points in $\mathcal{X} - \mathcal{T}$ will need to be mapped to points in $\mathcal{T}$, while some points in $\mathcal{U}$ will need to be mapped to points in $\mathcal{X} - \mathcal{T}$.

Grubbs, Ristenpart, and Yarom (GRY) first describe a solution (attributed to an anonymous reviewer) for this problem when $\mathcal{X}$ can be efficiently ranked. The solution uses the rank-encipher-unrank algorithm discussed earlier in the context of domain targeting, but with the ranking and unranking algorithms additionally performing a binary search through precomputed tables (which increase the space requirements) with the same size as the preservation set $\mathcal{T}$.

As explained above when discussing domain targeting, not all domains can be efficiently ranked, and it may otherwise be desirable to have solutions that avoid ranking. Additionally, GRY point out that the ranking solution could be more susceptible to timing attacks because of both the binary search and the ranking. Their main result is thus an algorithm called Zig-Zag that does not require $\mathcal{X}$ to be efficiently ranked, and only requires the ability to efficiently test membership in the preservation set $\mathcal{T}$. Much like Cycle Walking in the domain targeting setting, Zig-Zag has small expected running time but significantly higher worst-

case running time. Thus, Zig-Zag can be a good choice in practice, but has some of the same drawbacks as Cycle Walking. An interesting question, both for theory and for practice, is whether it is possible to do domain completion while avoiding an expected-time process and achieving a lower worst-case running time, while also avoiding ranking. Looking forward, our Cycle Slicer algorithm, when combined with some preprocessing on the table of mappings, will give a new algorithm for domain completion that avoids expected time and has better worst-case running time than Zig-Zag.

Our Main Result: The Cycle Slicer Algorithm. We now introduce our main result, an algorithm we call Cycle Slicer which gives new solutions to the problems introduced above. At a high level, one round of the Cycle Slicer algorithm transforms a permutation P on $\mathcal{X}$ into a matching on some subset of the points of $\mathcal{X}$, where by *matching* we mean a permutation made up of only transpositions (cycles of length 2). By carefully specifying which points should be included in the matching, formalized by an *inclusion function* I, we will be able to use Cycle Slicer to perform both domain targeting and domain completion.

To understand Cycle Slicer, consider the problem of transforming an arbitrary permutation P on $\mathcal{X}$ into another permutation on $\mathcal{X}$ that only consists of length 2 cycles, called transpositions or swaps. If we look at the cycle structure of P, we will likely find many cycles longer than length 2, some substantially longer. The main idea underlying Cycle Slicer is to "slice" up long cycles into a number of smaller, length 2 cycles. To do this, Cycle Slicer uses a *direction function*, *Dir*, which will simply be a function that gives a random bit for each point in $\mathcal{X}$. If the direction bit for a point x is 1 (i.e., $Dir(x) = 1$), then we say that x is forward-looking. If the direction bit is instead a 0, we call the point backward-looking. If a point x is forward-looking and the next point in the cycle, $P(x)$, is backward-looking, then those points are paired together. Similarly, if x is backward-looking and $P^{-1}(x)$ is forward-looking, then those points are paired up. It is easy to see that the use of the direction function pairs up some of the points in $\mathcal{X}$, while other points whose direction bits were not consistent with the direction bits of the points preceding and following them in their cycles are not paired up. Swapping any number of the paired points now results in a permutation on $\mathcal{X}$ made up of only transpositions.

As an example, suppose permutation P has a cycle $(w\ x\ y\ z)$. This means that $P(w) = x$, $P(x) = y$, $P(y) = z$, and $P(z) = w$. Now suppose our direction function gives bits 0, 1, 0, 0 for these points. This means that x is a forward-looking point, and the three others are all backward-looking. Based on this, x and y will be paired up which, if they are eventually swapped, will lead to a permutation with cycles $(w)(x\ y)(z)$.

The direction function determines which points are paired, but whether a pair is actually swapped also depends on another function we call the inclusion function. An inclusion function I takes two points as input and either outputs 1 or 0, with 0 meaning the points are not swapped. Looking forward, we will be able to apply Cycle Slicer to a number of different problems by specifying different inclusion functions.

One round of Cycle Slicer will result in many points swapping positions, but this alone will not sufficiently mix the points. Thus, for all of our applications we will need many rounds of Cycle Slicer. Since Cycle Slicer, like the Reverse Cycle Walking algorithm described above, mixes points through repeated random matchings, we follow the same approach as Miracle and Yilek for analyzing the number of rounds needed to yield a random permutation on the desired domain. They use techniques introduced by Czumaj and Kutylowski [7] to analyze a matching exchange process. At each step of a matching exchange process, a number k is chosen according to some distribution and then a matching of size is k is chosen uniformly. Both Reverse Cycle Walking and Cycle Slicer can be viewed as a matching exchange process. The analysis given by Czumaj and Kutylowski does not give explicit constants for their bounds, so in order to provide explicit constants for their Reverse Cycle Walking algorithm, Miracle and Yilek reprove several key lemmas from the Czumaj and Kutylowski result. We extend their work to also give explicit constants for general matching exchange processes based on two new parameters which we introduce. The first parameter is the probability that a specific pair of points (x, y) is in the matching and the second parameter is the probability that a second pair (z, w) is also in the matching conditioned on a first pair being in the matching. This analysis allows us to bound the variation distance in each of our applications, and we believe the parameters are general enough that our result is of interest beyond the application to the Cycle Slicer algorithm.

Now that we have described the basic Cycle Slicer algorithm, we will explain how it can be applied to domain targeting and completion.

Cycle Slicer and Domain Targeting. To use Cycle Slicer for domain targeting, constructing a permutation on $\mathcal{S} \subseteq \mathcal{X}$ out of permutations on $\mathcal{X}$, we simply define an inclusion function I_t that outputs 1 when given two points in the target set $\mathcal{S}$, and outputs 0 otherwise. This means that only points from $\mathcal{S}$ will be swapped. Repeating this process for a number of rounds results in a random permutation on the target set.

The resulting process is similar to Reverse Cycle Walking, but the important difference is that when the size of $\mathcal{S}$ is close to the size of $\mathcal{X}$, Cycle Slicer is still able to pair many points from $\mathcal{S}$ while Reverse Cycle Walking will struggle to pair points. This means substantially fewer rounds of Cycle Slicer are needed. In our running example with social security numbers where $|\mathcal{S}| = 10^9$ and $|\mathcal{X}| = 2^{30}$, Cycle Slicer will need about $1/2600$ as many rounds as Reverse Cycle Walking. We could improve the performance of RCW by instead letting $\mathcal{X}$ be a larger supserset of $\mathcal{S}$; if for RCW we let $\mathcal{X}$ be the set of 32-bit values, then its performance significantly increases and we have arguably a fairer comparison. But even with this enhancement for RCW, Cycle Slicer will still need only about $1/4$ as many rounds. We give a more detailed explanation of this in Sect. 5.

Cycle Slicer and Domain Completion. Using Cycle Slicer for domain completion is less straightforward, since the table of mappings already imposes constraints on the ultimate cycle structure of any permutation on the entire domain. In particular, the table of mappings can lead to a number of cycles and lines (se-

quences of points that do not form a cycle). The permutation we wish to build on the entire domain $\mathcal{X}$ needs to have a cycle structure that stays consistent with these. At a high level, our solution is to collapse the lines into a single point and use Cycle Slicer on the resulting induced domain. The "fake" points that represent entire lines from the table will be part of matchings with other points in the domain. After a number of rounds of Cycle Slicer, we can then substitute the entire line back in for the point that represented it. While this is the intuitive idea of what is happening, writing down an algorithm that allows for evaluation of a single point is non-trivial, and we consider this algorithm (found in Sect. 6) one of our main contributions.

This algorithm requires, for each point in a line, the ability to determine the first and last points in the line. This information can be precomputed and also made read-only along with the table. Given this precomputation, our algorithm has lower worst-case running time than the Zig-Zag construction, which could require $|\mathcal{T}|$ loop iterations in the worst case to encipher a single point. Our algorithm is also not expected-time, as enciphering any point outside of $\mathcal{T}$ simply uses the same r-round Cycle Slicer algorithm, and the number of rounds does not vary across different points. This could be of practical significance if reliable execution times are needed, or if there is concern about the potential for timing attacks.

Further Uses of Cycle Slicer. In addition to the applications of Cycle Slicer discussed above, we can imagine a number of other uses for the new algorithm. In particular, Cycle Slicer would seem to be useful in any situation where we want to use a permutation on some general domain to build a more specialized permutation on a related domain or that is consistent with some additional restrictions. Framed this way, our results are somewhat similar to the work of Naor and Reingold on constructing permutations with particular cycle structure [20]. In Sect. 7 we discuss three settings where we think Cycle Slicer could be useful.

2 Preliminaries

Notation. We denote by $x \leftarrow y$ the assignment of the value of y to x. If F is a function, then $y \leftarrow F(x)$ means evaluating the function on input x and assigning the output to y. When talking about a permutation P and its cycle structure, we use the notation $(\ldots\ x\ y\ \ldots)$ to indicate there is a cycle that contains x immediately followed by y, i.e., $P(x) = y$ and $P^{-1}(y) = x$. For any function F (which may or may not be a permutation), we say points $x_1, \ldots, x_j$ form a *line* if for all $1 \leq i \leq j-1$ it is true that $F(x_i) = x_{i+1}$.

Total Variation Distance. In order to generate a random permutation using the Cycle Slicer algorithm it is necessary to have many rounds of Cycle Slicer. To bound the number of rounds needed we will analyze the total variation distance. Let $x, y \in \Omega$, $P^r(x, y)$ be the probability of going from x to y in r steps and μ be a distribution on the state space Ω. Then the *total variation distance* is

defined as $||P^r - \mu|| = \max_{x \in \Omega} \frac{1}{2} \sum_{y \in \Omega} |P^r(x,y) - \mu(y)|$. In Sect. 4 we will bound the total variation distance between the distribution after r rounds of the Cycle Slicer algorithm and the uniform distribution.

3 The Cycle Slicer Construction

In this section we introduce our main contribution, the Cycle Slicer algorithm. As described in the introduction, at a high level, the Cycle Slicer algorithm allows us to transform any permutation on a set $\mathcal{X}$ into another permutation on $\mathcal{X}$ that may have additional desirable properties.

Formal Description. We now formally describe the Cycle Slicer algorithm. Let $P : \mathcal{X} \to \mathcal{X}$ be a permutation and $P^{-1} : \mathcal{X} \to \mathcal{X}$ be its inverse. Let $Dir : \mathcal{X} \to \{0,1\}$ be a function called the *direction function*, $I : \mathcal{X} \times \mathcal{X} \to \{0,1\}$ be a function called the *inclusion function*, and $B : \mathcal{X} \to \{0,1\}$ be a function called the *swap function*. Given these, we give pseudocode for one round of Cycle Slicer in Algorithm 1. This round of Cycle Slicer results in a permutation on $\mathcal{X}$, and is in fact an involution, serving as its own inverse algorithm. We will

Algorithm 1 Cycle Slicer (one round)

```
1: procedure CS_{P,P^{-1},I,Dir,B}(x)
2:     if Dir(x) = 1 then
3:         x' ← P(x)
4:         if Dir(x') = 0 and I(x, x') = 1 and B(x) = 1 then
5:             x ← x'
6:         end if
7:     else
8:         x' ← P^{-1}(x)
9:         if Dir(x') = 1 and I(x', x) = 1 and B(x') = 1 then
10:            x ← x'
11:        end if
12:    end if
13:    return x
14: end procedure
```

In words, to map a point $x \in \mathcal{X}$, Cycle Slicer first applies the direction function Dir to x to see if it should look forward ($Dir(x) = 1$) or backwards. If forward, then it applies the permutation P to x to get a point x' to potentially swap with. If backwards, then it instead applies the inverse permutation P^{-1} to get a point x'. Now, to decide whether or not to swap the positions of x and x', Cycle Slicer applies an inclusion function I and a swap function B. If both output 1, then x and x' are swapped. Note that our algorithm always lets the first input to I be the "forward-looking" point (i.e., the point x with $Dir(x) = 1$)

and, similarly, always applies B to the forward-looking point. This ensures we get consistent decision bits for both x and x'.

Algorithm 1 gives code for just one round of Cycle Slicer. In the applications later in the paper, we will need many rounds of Cycle Slicer to ensure points are sufficiently mixed. Towards this, for brevity we let CS^r denote r independent rounds of CS, each with an independent P, P^{-1}, Dir, and B. Each round of CS^r will use the same inclusion function I, however[3]. We then let CSinv^r denote the inverse of CS^r which, since CS is an involution, will just be the r rounds of CS^r applied in reverse order.

Looking forward to the applications of Cycle Slicer discussed in later sections, parameterizing Cycle Slicer with different decision functions I will lead to new solutions to those problems.

Discussion. Intuitively, Cycle Slicer gives us a way to "slice" up the cycles of the underlying permutation P and form a matching. More specifically, consider any point x in our domain $\mathcal{X}$. Suppose the permutation P puts x into a cycle $(\ldots\ w\ x\ y\ \ldots)$. We wish to construct a matching on $\mathcal{X}$, which is a permutation on $\mathcal{X}$ made up of only transpositions (2-cycles). To do this, we apply a direction function Dir to the points in $\mathcal{X}$. If $Dir(x) = 1$, we say x is forward-looking, which means it will potentially pair with $P(x) = y$. If $Dir(x) = 0$, we say it is backward-looking, which means it will potentially pair with $P^{-1}(x) = w$. Of course, the direction function is also being applied to w and y, so they will also be forward-looking or backward-looking. If x is forward-looking and y is backward-looking, they become a pair to potentially swap in a matching; similarly, if x is backward-looking and w is forward-looking, then x and w are a pair to potentially swap in a matching. Whether the points are actually swapped is determined by two other functions, the inclusion function I and the swap function B. The inclusion function allows us to restrict which points will be swapped based on properties like whether or not the two points are part of a particular subset of $\mathcal{X}$. The swap function will simply be a bit flip to determine if the swap should occur or not, and is needed for technical reasons.

Note that at a high-level our analysis of the Cycle Slicer algorithm relies on viewing the algorithm as a Markov chain on the set of permutations of some set $\mathcal{V} \subset \mathcal{X}$ where at each step an independent matching on $\mathcal{V}$ is applied. The independence requirements for P, P^{-1}, B and Dir at each round ensure that an independent matching is applied at each step which is needed to apply the techniques from Czumaj and Kutylowski's analysis which we do in Sect. 4 and Sect. A. The inclusion function I will be used to determine which pairs in the matching are allowed to ensure we generate a matching on the correct set.

Practical Considerations. There are a number of ways to instantiate the different components of Cycle Slicer in practice, and the specific algorithms used would obviously depend on the application. The most obvious instantiation

[3] It is possible there will be other interesting uses of Cycle Slicer that use different inclusion functions in different rounds. However, our applications do not need this, so to keep things simpler we just use a single inclusion function across all rounds.

would use a block cipher $E : \mathcal{K} \times \mathcal{X} \to \mathcal{X}$ for the round permutation P (and the inverse block cipher E^{-1} for P^{-1}), with each round requiring a different block cipher key. Yet, we will soon see that for typical applications we will need thousands of rounds of Cycle Slicer, so a separate key for each round is not ideal. Instead, it would make sense to use a tweakable block cipher [14] with a single key and the round number used as a tweak. Specifically, to evaluate the ith round permutation P_i on a point $x \in \mathcal{X}$, we would compute $E(K, i, x)$, where K is the randomly chosen block cipher key. Since our proposed applications are all in the area of format-preserving encryption, E might be instantiated with one of the standardized Feistel-based modes [3, 5, 9] (though, based on recent attacks [1, 8], care should be taken) or, if stronger provable-security guarantees are desired, perhaps the Swap-or-Not cipher [12] or one of the fully-secure ciphers from [18, 21]. All of these options support tweaks.

The direction functions and swap functions could then be instantiated with a pseudorandom function (PRF); a single key could be used for each if the round number is included as an input to the PRFs. Specifically, if we are performing r rounds of Cycle Slicer and have a PRF $F : \mathcal{K}' \times \{1, \ldots, r\} \times \mathcal{X} \to \{0, 1\}$, we first choose a random key K_1 for the direction functions and a random key K_2 for the swap functions. Then, to evaluate the direction function (resp. swap function) in round j on a point x, we would compute $F(K_1, j, x)$ (resp. $F(K_2, j, x)$).

Looking Forward: Analysis of Cycle Slicer. In the next section, we give an analysis of Cycle Slicer, proving an information theoretic, mixing-time result. More specifically, we give a bound on total variation distance after many rounds of Cycle Slicer, with each round's permutation, direction function, and swap function chosen randomly from the appropriate sets of such functions. We emphasize that this makes our results very modular and applicable to a number of different problems, since typical security proofs in our target application areas first swap out computationally secure components like block ciphers and PRFs with randomly chosen permutations and functions, respectively.

4 Analyzing Cycle Slicer

In order to bound the number of rounds of the Cycle Slicer algorithm that are needed, we will analyze a more general process that generates a permutation on a set of points by applying a matching at each step. Specifically we will analyze a type of matching exchange process first defined and analyzed by Czumaj and Kutylowski [7]. A *matching exchange process* is a Markov chain for generating a permutation on a set of n elements. At every step a number $\kappa \leq n/2$ is selected according to some distribution and a matching of size κ is chosen uniformly at random. Independently, for each pair (x, y) of the matching, with probability $1/2$ the elements x and y are exchanged. Czumaj and Kutylowski [7] prove that as long as the expected value of κ is constant then after $\Theta(\log(n))$ steps, the variation distance is $O(1/n)$. We can view the Cycle Slicer algorithm as a matching exchange process and it can be shown that the expected size of a matching is constant. However, the result of [7] does not give explicit bounds

on the constants. Miracle and Yilek [17] extend the result of [7] by reproving several key lemmas to give explicit constants for a particular algorithm. Here we extend their work to a more general setting. Given a matching exchange process where the matchings are selected according to the following parameters, we provide explicit bounds, including constants, on the mixing time and variation distance of the matching exchange process. In Sects. 5, 6 and 7 we will show how our analysis can be applied to the Cycle Slicer algorithm in our three different applications. We begin by defining the parameters we will need.

1. For any points x, y the probability that a pair (x, y) is part of a matching is at least p_1.
2. For any points x, y, z, and w conditioned on (x, y) being a pair in the matching, the probability that (z, w) is also in the matching is at least p_2.

Note that p_1 and p_2 refer to the probability these pairs are included in the generated matching (i.e., the matching generated by Cycle Slicer) regardless of the result of the bit flip (or swap function) that determines whether the points are actually flipped. Assuming p_1 and p_2 are $\Omega(1/n)$, we will show the process mixes in time $O(\log n)$. Formally, the *mixing time* of a Markov chain $\mathcal{M}$ with state space Ω is defined as $\tau_{\mathcal{M}}(\epsilon) = \min\{t : ||\mathcal{P}^{t'} - \mu|| \leq \epsilon, \forall t' \geq t\}$ where $||\mathcal{P}^{t'} - \mu|| = \max_{x \in \Omega} \frac{1}{2} \sum_{y \in \Omega} |\mathcal{P}^{t'}(x, y) - \mu(y)|$, $\mathcal{P}^{t'}(x, y)$ is the probability of going from x to y in t steps and μ is the stationary distribution of $\mathcal{M}$. We prove the following result on the mixing time of the matching exchange process with parameters p_1 and p_2 as defined above.

Theorem 1. *For* $T \geq \max\left(40\ln(2n^2), \frac{10\ln(n/9)}{\ln(1+p_1p_2(7/36)((7/9)n^2-n))}\right) + \frac{72\ln(2n^2)}{p_1 n}$ *the mixing time* τ *of a matching exchange process with parameters* p_1 *and* p_2 *as defined above satisfies*

$$\tau(\epsilon) \leq T \cdot \left\lceil \frac{\ln(n/\epsilon)}{\ln n^2} \right\rceil.$$

When $\epsilon = 1/n$ *and* $p_1, p_2 = \Omega(1/n)$*, the bound simplifies to* $\tau(1/n) = \Theta(\ln(n))$.

The proof of Theorem 1 can be found in Appendix A. In order to analyze our Cycle Slicer algorithm a bound on the variation distance will be more useful. This is obtained by a straightforward manipulation of the mixing time bound above. As long as $p_1, p_2 = \Omega(1/n)$ and the number of rounds is at least $T = \Theta(\ln(n))$, the variation distance is less than $1/n$. Let ME^r represent r rounds of the matching exchange (ME) process and ν_{ME^r} be the distribution on permutations of n elements after r rounds of the matching exchange process. More specifically, $\nu_{\mathrm{ME}^r}(x, y)$ is the probability of starting from permutation x and ending in permutation y after r rounds of the matching exchange process. Applying the definition from Sect. 2 gives us $||\nu_{\mathrm{ME}^r} - \mu_s|| = \frac{1}{2} \sum_{y \in \Omega} |\nu_{\mathrm{ME}^r}(x, y) - \mu_s(y)|$ where Ω is the set of all permutations on n elements and μ_s is the uniform distribution Ω. Using this definition and Theorem 1 we have the following.

Corollary 1. *Let* $T = \max\left(40\ln(2n^2), \frac{10\ln(n/9)}{\ln(1+p_1p_2(7/36)((7/9)n^2-n))}\right) + \frac{72\ln(2n^2)}{p_1 n}$, *then*

$$||\nu_{ME^r} - \mu_s|| \leq n^{1-2r/T},$$

where ν_{ME^r} *is the distribution after* r *rounds of the matching exchange process and* μ_s *is the uniform distribution on permutations of the* n *elements.*

Cycle Slicer. Next, we use Corollary 1 to bound the total variation distance for the Cycle Slicer algorithm. In the following sections we will look at the Cycle Slicer algorithm in several different applications. In each of these we are effectively generating a permutation on a set $\mathcal{V}$ which is a subset of a larger set $\mathcal{X}$. At each step of Cycle Slicer we use a permutation P on the points in $\mathcal{X}$ to generate a matching on the points in $\mathcal{V}$. Here we prove a general result for this setting. Let $\mathcal{V} \subset \mathcal{X}$ and the inclusion function I be defined as $I(x, y) = 1$ if and only if $x \in \mathcal{V}$ and $y \in \mathcal{V}$. Note that the permutation P chosen at each round is a uniformly random permutation on $\mathcal{X}$ and B and Dir are as defined in Sect. 3 (i.e. independent random bits). In this setting, we prove the following theorem.

Theorem 2. *Let* $T = \max\left(40\ln(2|\mathcal{V}|^2), \frac{10\ln(|\mathcal{V}|/9)}{\ln(1+(7/144)((7/9)|\mathcal{V}|^2-|\mathcal{V}|)/|\mathcal{X}|^2)}\right)$ $+\frac{144|\mathcal{X}|\ln(2|\mathcal{V}|^2)}{|\mathcal{V}|}$, *then*

$$||\nu_{\mathsf{CS}^r} - \mu_s|| \leq |\mathcal{V}|^{1-2r/T},$$

where ν_{CS^r} *is the distribution after* r *rounds of* CS *and* μ_s *is the uniform distribution on permutations on* $\mathcal{V}$.

Note that if $|\mathcal{V}|$ is a constant fraction of $|\mathcal{X}|$ then as long as the number of rounds is at least $\Theta(\ln(|\mathcal{V}|))$ then the variation distance will be less than $1/|\mathcal{V}|$.

Proof. In order to apply Corollary 1 to the Cycle Slicer algorithm, we need to bound the parameters p_1 and p_2. Recall that for any pair of points (x, y), p_1 is the probability that the pair (x, y) is part of a potential matching. At any step of Cycle Slicer, for any points $x, y \in \mathcal{V}$, (x, y) is part of the matching if one of two things happen, either $Dir(x) = 1$, $Dir(y) = 0$ and $P(x) = y$ or $Dir(x) = 1$, $Dir(y) = 0$, and $P^{-1}(x) = y$. Each of these events happens with probability $(1/2)(1/2)(1/|\mathcal{X}|)$, implying that $p_1 = (2|\mathcal{X}|)^{-1}$. Note that it is also required that $I(x, y) = 1$. However, this is always true since $x, y \in \mathcal{V}$ and $I(x, y) = 1$ if and only if $x, y \in \mathcal{V}$.

For any points $x, y, z, w \in \mathcal{V}$ conditioned on (x, y) being a pair in the matching, p_2 is the probability that (z, w) is also in the matching. Again, there are two situation where this can happen. Without loss of generality we will assume that $Dir(x) = 1, Dir(y) = 0$ and $P(x) = y$. The other case ($Dir(x) = 0, Dir(y) = 1$ and $P^{-1}(x) = y$) is almost identical. The pair (z, w) will be in a matching if $Dir(z) = 1$, $Dir(w) = 0$ and $P(z) = w$. There are a total of $|\mathcal{X}| - 1$ points that z could get mapped too (we already know z is not mapped to y since (x, y) is in the matching) so this case happens with probability $(1/2)(1/2)(1/(|\mathcal{X}| - 1))$. Similarly, the pair (z, w) will be in the matching if $G_i(z) = 0$, $G_i(w) = 1$ and

$\mathsf{P}_i^{-1}(z) = w$ and this occurs with probability $(1/2)(1/2)(1/(|\mathcal{X}|-1))$. Combining these shows $p_2 = (2(|\mathcal{X}|-1))^{-1} \geq (2|\mathcal{X}|)^{-1}$. Using these bounds on p_1 and p_2 we can now directly apply Corollary 1 to obtain the above theorem.

□

5 Domain Targeting

BACKGROUND. Our first application of the Cycle Slicer algorithm is in what we are calling *domain targeting*. In domain targeting, we wish to construct an efficiently-computable permutation on a target set $\mathcal{S}$. Yet, this target set might have "strange" structure, and it might not be clear how to directly build a permutation on such a set.

As described in the introduction, this problem can arise in format-preserving encryption, and a well-known strategy for solving it is to take a permutation P on a larger, "less strange" set $\mathcal{X}$ for which $\mathcal{S} \subseteq \mathcal{X}$, and then transform a permutation on $\mathcal{X}$ into a permutation on the target set $\mathcal{S}$. One such transformation is known as *Cycle Walking*, in which the permutation P on the larger set $\mathcal{X}$ is repeatedly applied to a point $x \in \mathcal{S}$ until the output of the permutation finally lands back in the target set $\mathcal{S}$. This leads to a small *expected* running time, but the worst-case running time is high and, additionally, enciphering different points may take different amounts of time.

More recently, Miracle and Yilek [17] introduced an alternative to Cycle Walking called *Reverse Cycle Walking* (RCW). RCW uses permutations on the larger set $\mathcal{X}$ to construct matchings on the target set $\mathcal{S}$. In a round of RCW using permutation P on $\mathcal{X}$, if two consecutive points x, x' in a cycle are both in $\mathcal{S}$ and are immediately preceded and followed by points *not* in $\mathcal{S}$, then x and x' are paired and, depending on a bit flip, either swapped or not. Said another way, $x, x' \in \mathcal{S}$ are only potentially swapped in a round of RCW if they are part of a cycle $(\ldots\, y\ x\ x'\ z \ldots)$ and $y, z \notin \mathcal{S}$.

As a result, in every round of RCW some fraction of the points in the target set $\mathcal{S}$ are paired and swapped. Repeating this process for many rounds eventually mixes the points in $\mathcal{S}$ sufficiently. RCW has lower worst-case running time than regular Cycle Walking, and the time to encipher a point does not vary depending on the input. As we will see, Cycle Slicer, like RCW, will give us a way to form a matching on the points in $\mathcal{S}$ and, with enough rounds, sufficiently mix the points.

The main result in this section is that the Cycle Slicer algorithm can be used to improve upon Reverse Cycle Walking in the important scenario when the size of the target set is a large constant fraction of the size of the larger set $\mathcal{X}$. A simple example of such a situation would be if $\mathcal{S}$ is the set of bitstrings that give valid 9 digit decimal numbers and $\mathcal{X}$ is the set of 30 bit strings. In this case, $|\mathcal{S}| = 10^9$, $|\mathcal{X}| = 2^{30}$, and the ratio $10^9/2^{30} \approx .93$. In such scenarios, which are important in practice, Cycle Slicer will require significantly less rounds than RCW.

To understand why this will be the case, consider a scenario in which the size of the target set $\mathcal{S}$ is a large fraction of the size of $\mathcal{X}$. A random permutation P on $\mathcal{X}$ will likely yield many cycles with points mostly from $\mathcal{S}$ and very few points from $\mathcal{X} - \mathcal{S}$. For example, we could have a cycle $(x\ x'\ y\ x''\ x''')$ in which only y is not in the target set and all of the x's are in $\mathcal{S}$. With Reverse Cycle Walking, none of the x's would be swapped, while with Cycle Slicer there will be a reasonable probability some or even all of the x's will be swapped, contributing to the mixing.

Details. More formally, suppose we wish to construct a permutation on a set $\mathcal{S}$ which is a subset of a larger set $\mathcal{X}$ for which we already know how to construct efficient permutations. Then we can use Cycle Slicer with the following inclusion function I_t:

$$I_t(x, y) = \begin{cases} 1 \text{ If } x \in \mathcal{S} \text{ and } y \in \mathcal{S} \\ 0 \text{ Otherwise} \end{cases}$$

In words, we only include two points in a potential swap with the Cycle Slicer algorithm if both points are in the target set $\mathcal{S}$.

Finally, we will use our result from Sect. 4 to bound the number of steps of the Cycle Slicer algorithm needed for domain targeting. We will use Theorem 2 which bounds the variation distance of the Cycle Slicer algorithm in terms of the size of the domain $\mathcal{X}$ and the number of points in the set we are generating a permutation on $\mathcal{V}$. In domain targeting we are generating a permutation on $\mathcal{S}$ with size $|\mathcal{S}|$. This directly gives the following corollary to Theorem 2.

Corollary 2. *Let* $T = \max\left(40\ln(2|\mathcal{S}|^2), \frac{10\ln(|\mathcal{S}|/9)}{\ln(1+(7/144)((7/9)|\mathcal{S}|^2-|\mathcal{S}|)/|\mathcal{X}|^2)}\right) + \frac{144|\mathcal{X}|\ln(2|\mathcal{S}|^2)}{|\mathcal{S}|}$, *then*

$$||\nu_{\mathsf{CS}^r} - \mu_s|| \leq n^{1-2r/T},$$

where ν_{CS^r} *is the distribution after* r *rounds of* CS *and* μ_s *is the uniform distribution on permutations on* $\mathcal{S}$.

Comparison with RCW. It is difficult to see from the theorem above but our algorithm gives a significant advantage over RCW in the very typical case where the size of the target set is at least half the size of the domain (i.e. $|\mathcal{S}| \geq |\mathcal{X}|/2$). For example, consider the setting of encrypting social security numbers using an existing cipher for 30 bit strings. Here the size of the target set (i.e., the number of 9 digit numbers) is 10^9 and the domain has size 2^{30}. In this setting our algorithm requires 12,257 rounds until the variation distance is less than $1/10^9$ while RCW requires over 32 million. While these numbers make RCW look completely impractical, it is important to note that since RCW works best when the size of the target set is between 1/4 and 1/2 the size of $\mathcal{X}$, in some settings it may be possible to find a larger superset of the target set to use with RCW. If using RCW in our social security number example, it would actually make more sense to choose $\mathcal{X}$ to be the set of 32 bit values instead of 30 bit values. In this case, RCW requires just over 44,000 rounds. Nevertheless, Cycle Slicer is still significantly faster, requiring around 1/4 the number of rounds.

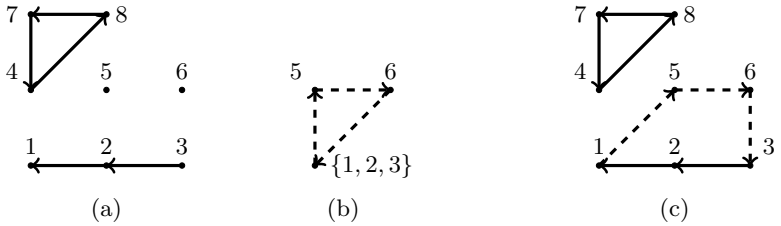

Fig. 1. Cycle Decomposition for $\mathcal{T} = \{2, 3, 4, 7, 8\}$, $\mathcal{U} = \{1, 2, 4, 7, 8\}$ with mappings $3 \to 2, 2 \to 1, 4 \to 8, 8 \to 7$ and $7 \to 4$

6 Domain Completion

Background. Our second application of Cycle Slicer is in *domain completion*. This problem, recently studied by Grubbs, Ristenpart, and Yarom [11], arises when we wish to construct a permutation on some set $\mathcal{X}$, but we need the permutation to maintain some mappings we might already have stored in a table. More formally, let $\mathcal{X}$ be the set we wish to construct a permutation on. Let $\mathcal{T} \subseteq \mathcal{X}$ be called the preservation set, which will be the set of domain points for which we already have a mapping. This mapping is given by a 1-1 and onto function $G : \mathcal{T} \to \mathcal{U}$ with $\mathcal{T}$,$\mathcal{U} \subseteq \mathcal{X}$. The domain completion problem is then to construct an efficiently computable permutation $F : \mathcal{X} \to \mathcal{X}$ (with a corresponding efficiently computable inverse function F^{-1}) such that $F(x) = G(x)$ for all $x \in \mathcal{T}$.

To understand our algorithm, note that the cycle decomposition (or cycle structure) of the partial permutation on $\mathcal{X}$ given by the mapping G consists of a set of cycles, lines and single points (see Fig. 1). The cycles and lines come from the mapping G while the single points are the points in $\mathcal{X} - \mathcal{T} - \mathcal{U}$. At a high-level our algorithm ignores any cycles, collapses any lines to a single point and generates a matching on the remaining set of points using the Cycle Slicer algorithm. When the previously collapsed lines are expanded and the cycles are added back in, this gives a permutation on $\mathcal{X}$ that preserves the mappings given by G. Figure 1 gives an example of this process. Figure 1(b) gives a permutation on the collapsed line $3 \to 2 \to 1$ and single points 5 and 6. Figure 1(b) shows how to expand the line and create a permutation on the whole set. To implement this idea we will use procedure CSDC in Algorithm 2 which performs some preprocessing before using Cycle Slicer as a subroutine. We also describe the inverse of this procedure CSDCinv in Algorithm 2.

Details. We will use the Cycle Slicer algorithm as defined in Sect. 3 (Algorithm 1) with the following inclusion function I_c:

$$I_c(x, y) = \begin{cases} 1 \text{ If } x \notin \mathcal{U} \text{ and } y \notin \mathcal{U} \\ 0 \text{ Otherwise} \end{cases}$$

Algorithm 2 Domain Completion Using Cycle Slicer

```
1: procedure CSDC(x)
2:     if x ∈ T then
3:         x ← G(x)
4:     else
5:         if x ∈ U then
6:             x ← frst(x)
7:         end if
8:         x ← CS^r(x)
9:     end if
10:    return x
11: end procedure
12: procedure CSDCinv(x)
13:    if x ∈ U then
14:        x ← G^{-1}(x)
15:    else
16:        x ← CSinv^r(x)
17:        if x ∈ T then
18:            x ← lst(x)
19:        end if
20:    end if
21:    return x
22: end procedure
```

Next, we will describe the procedures *frst* and *lst* used by CSDC and CSDCinv respectively. These are again easiest to understand by considering the cycle decomposition of the mapping G. Notice that the function *frst* is only applied to points x such that $x \in \mathcal{U} - \mathcal{T}$ which implies that in the cycle decomposition x is the last point in a line. We will define $frst(x)$ to be the first point in the line containing x. Note that $frst(x) \in \mathcal{T} - \mathcal{U}$. For example, in Fig. 1 $frst(1) = 3$. Similarly, the function *lst* is only applied to points at the beginning of a line (i.e., a point $x \in \mathcal{T} - \mathcal{U}$). We will define $lst(x)$ to be the last point in the line containing x (again in Fig. 1 $lst(3) = 1$). If the functions *frst* and *lst* are not available they can either be precomputed or computed on the fly. Algorithm 3 gives the detailed procedures for computing *frst* and *lst*. Note that in the worst case they each take time $O(|\mathcal{T}|)$. However, *frst* only needs to be computed for points $x \in \mathcal{U} - \mathcal{T}$ and *lst* only needs to be computed for points $y \in \mathcal{T} - \mathcal{U}$. This implies that if only the necessary values of *frst* and *lst* are precomputed then the overall running time will be $O(|\mathcal{T}|)$ because the algorithm will only traverse each line in the cycle decomposition twice (once for *frst* of the endpoint and once for *lst* of the start point).

Correctness of CSDC. Next, we will prove that the CSDC algorithm generates a valid permutation F on $\mathcal{X}$ consistent with the mappings given by G. Notice that if we did not use the preprocessing algorithm given by CSDC but instead

Algorithm 3 Computing *frst* and *lst*

```
1: procedure frst(x)
2:     while x ∈ 𝒰 do
3:         x ← G^{-1}(x)
4:     end while
5:     return x
6: end procedure
7: procedure lst(x)
8:     while x ∈ 𝒯 do
9:         x ← G(x)
10:    end while
11:    return x
12: end procedure
```

simply applied the Cycle Slicer algorithm using the inclusion function I_c to the points in $\mathcal{X} - \mathcal{U}$, after enough steps we would generate a random permutation on the set $\mathcal{X} - \mathcal{U}$. This is almost a valid permutation except that it does not give us a mapping for the last point in every line (i.e. points in $\mathcal{U} - \mathcal{T}$) and it does give a mapping for the first point in every line (i.e. points in $\mathcal{T} - \mathcal{U}$) which are already mapped by G. Again, consider the cycle decomposition of G. Each line in this partial cycle decomposition contains exactly one point in $\mathcal{T} - \mathcal{U}$ namely the first point in the line and one point in $\mathcal{U} - \mathcal{T}$ namely the last point in the line. The preprocessing done by CSDC fixes this by first mapping a point $x \in \mathcal{U} - \mathcal{T}$ to the first point in the line containing x (i.e. $\mathit{frst}(x)$) which is in $\mathcal{T} - \mathcal{U}$ and by using the existing mapping in G for points in $\mathcal{T} - \mathcal{U}$. For the remaining points (i.e., $x \in \mathcal{X} - \mathcal{T} - \mathcal{U}$) no preprocessing is done and the Cycle Slicer algorithm is applied directly. It is straightforward to see from the cycle decomposition view of G that *frst* and *lst* give a bijection between the points in $\mathcal{T} - \mathcal{U}$ and the points in $\mathcal{U} - \mathcal{T}$. Thus if Cycle Slicer gives a random permutation on $\mathcal{X} - \mathcal{U}$ than the addition of our preprocessing algorithm CSDC gives a mapping that is uniformly random over all possible ways to extend the mapping G to a permutation F on $\mathcal{X}$.

Finally, we will use our result from Sect. 4 to bound the number of steps of the Cycle Slicer algorithm needed in our domain completion algorithm. We have shown above that our algorithm can be viewed as first using Cycle Slicer to generate a permutation on the set $\mathcal{X} - \mathcal{U}$ which is then mapped bijectively to a permutation on the set $\mathcal{X}$ which is consistent with the mapping G. For simplicity, in the analysis of the Cycle Slicer portion of the algorithm, we view the algorithm in this context as generating a permutation on the set $\mathcal{X} - \mathcal{U}$. Through the bijection the analysis then applies directly to permutations on the set $\mathcal{X}$ consistent with the mapping G. Here however the variation distance is the distance between the distribution after r rounds and the uniform distribution on all permutations consistent with G (instead of uniform on all permutations of $\mathcal{X} - \mathcal{U}$).

Theorem 2 from Sect. 4 bounds the variation distance of the Cycle Slicer algorithm and applies directly to the domain completion setting. In this application, the set $\mathcal{V}$ that we are generating a permutation on has size $|\mathcal{X} - \mathcal{U}|$, which gives the following corollary to Theorem 2.

Corollary 3. *Let* $T = \max\left(40\ln(2|\mathcal{Y}|^2), \frac{10\ln(|\mathcal{Y}|/9)}{\ln(1+(7/144)((7/9)|\mathcal{Y}|^2-|\mathcal{Y}|)/|\mathcal{X}|^2)}\right)$ $+\frac{144|\mathcal{X}|\ln(2|\mathcal{Y}|^2)}{|\mathcal{Y}|}$, *then*

$$||\nu_{\mathsf{CS}^r} - \mu_s|| \leq n^{1-2r/T},$$

where $\mathcal{Y} = \mathcal{X} - \mathcal{U}$, ν_{CS^r} *is the distribution after* r *rounds of* CS *and* μ_s *is the uniform distribution on permutations on* $\mathcal{X}$ *consistent with* G.

Comparison to Zig-Zag. As we stated in the introduction, one of our goals with Cycle Slicer in domain completion is to improve on the worst-case running time of the Zig-Zag construction from [11], which has a loop that repeats $|\mathcal{T}|$ times in the worst-case. (In each loop iteration, Zig-Zag additionally needs to check membership in $\mathcal{T}$, evaluate the underlying permutation, and do a table look-up.) Additionally, we want to avoid an expected-time procedure to minimize leaked timing information.[4]

At first glance, since CSDC and CSDCinv rely on the *frst* and *lst* functions, it would appear that we also have an expected-time procedure. But, the crucial difference is that we can precompute *frst* and *lst* for each point in the table. This precomputation can either be done (inefficiently) using the procedures in Algorithm 3, or more efficiently as described earlier in this section.

The new table with this precomputed information for each point can still be made read-only, and then evaluating *frst* and *lst* in our CSDC and CSDCinv algorithms will become a simple table look-up. This means the worst-case running time will simply be tied to the number of rounds needed for Cycle Slicer, which will be on the order of $\log|\mathcal{X}|$. Using our running example with social security numbers, if the preservation set has size 1 million (meaning that before we started using a FPE scheme, we manually encrypted 1 million customer's SSNs through lazy sampling and put them in the table), then in the worst-case Zig-Zag needs 1 million loop iterations, while the bounds in our Cycle Slicer results tell us we would need around 11,000 rounds to get a strong level of security. Further, after the precomputation, our algorithms CSDC and CSDCinv will need this many rounds for any point outside of the preservation set, so the running time will not vary widely based on which point is being enciphered.

[4] GRY argue that Zig-Zag does not leak damaging timing information, even though enciphering different points may result in very different execution times. We believe there could still be timing attacks in certain applications. For example, if an adversary measures execution time and then learns the corresponding plaintext, and then later observes a different execution time, then the adversary knows the same point was not enciphered this second time, which may or may not be useful and depends on the application.

Comparison to Ranking Solution. The ranking-based construction described by GRY, like our Cycle Slicer construction, requires extra storage in addition to the preservation set table. In particular, it requires computing the rank of each element of $\mathcal{T}$ and $\mathcal{U}$. These two lists of ranks then need to be sorted. Enciphering a point involves applying rank-encipher-unrank, but with the addition of performing a binary search on the sorted lists of ranks. However, this $\log |\mathcal{T}|$ factor is still less than the 11,000 rounds we need with Cycle Slicer in the above example.

Nevertheless, our construction does potentially have some advantages over the ranking-based construction. First, as we have already stated, our results do not rely on the ability to efficiently rank the desired domain, which can be important for non-regular languages or even regular languages described by a particularly complicated regular expression. Second, our construction could be more resistant to timing attacks. Specifically, GRY explain that implementations of both the binary search and the ranking/unranking[5] could leak important timing information. Our construction, on the other hand, always applies the same number of rounds of Cycle Slicer to points outside of the preservation set, which should be easier to implement without leaking timing information.

7 More Applications of Cycle Slicer

As we mentioned in the introduction, we believe Cycle Slicer will prove useful for any problem requiring the construction of permutations on domains with additional restrictions or constraints. Towards this, in this section we briefly discuss three problems where Cycle Slicer could be applied. We leave further investigation to future work.

Domain Extension. In Sect. 6 we saw that Cycle Slicer is useful for solving the problem of domain completion. Here we argue it would also work well for the closely-related *domain extension* problem, also recently investigated by Grubbs, Ristenpart, and Yarom [11].

Formally, let $\mathcal{D}$ be a set for which we already have some mappings. This means there is a preservation set $\mathcal{T} \subseteq \mathcal{D}$ and a 1-1 and onto function $G : \mathcal{T} \to \mathcal{U}$ with $\mathcal{T}$,$\mathcal{U} \subseteq \mathcal{D}$. The domain extension problem is then to construct an efficiently computable permutation $P : \mathcal{X} \to \mathcal{X}$ on a *larger* set $\mathcal{X} \supseteq \mathcal{D}$ while maintaining the property that $P(x) = G(x)$ for all $x \in \mathcal{T}$. Contrast this with the domain completion problem, in which we would only be interested in constructing a permutation on $\mathcal{D}$. As GRY observe, we cannot hope to construct a random permutation on the larger domain $\mathcal{X}$, since points in the table are all randomly mapped into the smaller domain $\mathcal{D}$, which would be very unlikely in a random permutation on $\mathcal{X}$. Nevertheless, the same algorithm we presented in Sect. 6 would work for this new problem, and multiple rounds would mix the points outside of the table as well as possible.

[5] For example, many ranking algorithms use a procedure similar to cycle walking.

Partitioned Domains. Another application of Cycle Slicer would be in a situation where we need a permutation on a domain $\mathcal{X}$ that can be partitioned into k smaller domains $\mathcal{X}_1, \ldots, \mathcal{X}_k$, and we want the permutation to map points in $\mathcal{X}_i$ to $\mathcal{X}_i$ for all i. For example, we might want a permutation on n bit strings that maps each point to another point with the same first three bits. To apply Cycle Slicer to this problem, we would let the inclusion function allow the swap of two points only if they come from the same domain $\mathcal{X}_i$.

Programmed Identity Mappings. One last application we will mention is in constructing permutations with some identity mappings that we may want "programmed". For example, we may want points $x, y, z \in \mathcal{X}$ to all be mapped to themselves, while all other points should be randomly mixed by the permutation. This is essentially a special case of domain completion, so our techniques in Sect. 6 would apply.

Acknowledgements. We thank the anonymous ASIACRYPT reviewers for providing detailed feedback that really helped improve the paper.

A Proof of Mixing Time Theorem

Here we provide the details of the proof of Theorem 1 which bounds the mixing time of a matching exchange process with parameters p_1 and p_2. Recall that p_1 is the probability a particular pair (x, y) is in the matching and p_2 is the probability that conditioned on one pair being in the matching, a second pair is also in the matching (this is formally defined in Sect. 4). Our proof relies heavily on the techniques used by Czumaj and Kutylowski [7] and the work by Miracle and Yilek [17] in the context of the Reverse Cycle Walking algorithm. We use the same techniques and thus our main contribution is the definition of the parameters p_1 and p_2 and the determination of explicit constants in terms of p_1 and p_2 which apply to more general matching exchanges processes rather than the specific Reverse Cycle Walking algorithm given in [17]. The proof of Czumaj and Kutylowski [7] gives a result on the mixing time of matching exchange processes that we can directly apply to our settings however their result is an asymptotic result and they do not provide explicit constant. Similar to Miracle and Yilek [17], we provide explicit constants in terms of p_1 and p_2 by reproving several key lemmas. For completeness we give a brief overview of the entire proof here and describe in details the two lemmas we have modified. The complete details of the proof can be found in [17].

As in [17] we will use the delayed path coupling theorem introduced by Czumaj and Kutylowski which builds on the techniques of coupling and path coupling both of which are commonly used techniques in the Markov chain community (see e.g., [13], [22]). A *coupling* of a Markov chain with state space Ω is a joint Markov process on $\Omega \times \Omega$ where Ω is the state space. A coupling requires that the marginal probabilities agree with the Markov chain probabilities and

that once the processes collide they move together. The expected time until the two processes collide gives a bound on the mixing time. We will let the distance d between two configurations be defined as the minimum number of transpositions (or exchanges of two elements) needed to transition from one configuration to the other. Using delayed path coupling, like with path coupling, we can restrict our attention to pairs of configurations that initially differ by a single transposition (pairs at distance one) and show that with sufficiently high probability after a logarithmic number of steps of a matching exchanges processes, the two processes will have collided.

Theorem 3. *Let d be a distance metric defined on $\Omega \times \Omega$ which takes values in $\{0, \ldots, D\}$, let $U = \{(x, y) \in \Omega \times \Omega : d(x, y) = 1\}$ and let δ be a positive integer. Let $(x_t, y_t)_{t \in \mathbb{N}}$ be a coupling for $\mathcal{M}$, such that for every $(x_{t\delta}, y_{t\delta}) \in U$ it holds that $\mathbf{E}\left[\, d(x_{(t+1)\delta}, y_{(t+1)\delta}) \,\right] \leq \beta$ for some real $\beta < 1$. Then,*

$$\tau_{\mathcal{M}}(\epsilon) \leq \delta \cdot \left\lceil \frac{\ln(D * \epsilon^{-1})}{\ln \beta^{-1}} \right\rceil.$$

As in the result of Miracle and Yilek we will use the same exact coupling introduced by Czumaj and Kutylowski. We provide a very brief description here but a more thorough one can be found in [7] and [17]. For one process we will matchings $m_1, m_2, \ldots$ that are selected completely randomly according to our matching exchange process probabilities. However for the second process we will use matchings $n_1, n_2, \ldots$ that are very closely related to the first set of matchings $m_1, m_2, \ldots$. Consider a pair of configurations that differ by a single inversion (x, y). If the exact same inversion is selected as a pair in the first matching m_1 then it is possible to couple the processes in one step by using the same matching for both processes but different bit flips for the pair (x, y). However this happens with probability p_1 which in our applications is only $\Theta(1/n)$ and does not give a tight enough bound. Czumaj and Kutylowski observed that it is likely that both x and y will be paired with different elements (x, z) and (y, w). If (x, y) are paired in the next matching m_2 they will choose $n_1 = m_1 - (x, z) - (y, w) + (y, z) + (x, w)$ and then different bit flips for (x, y) in m_2 and n_2. If (z, w) are paired in the next matching m_2 then they will choose $m_1 = n_1$ and then different bit flips for (x, y) in m_2 and n_2. In either case the two processes will have coupled (or collided) after 2 steps. Czumaj and Kutylowski refer to these pairs (x, y) and (y, z) as good pairs and they show that at each step the set of good pairs doubles and after $\Omega(\log(n))$ steps there are a linear number of these and thus it is very likely that a good pair will be an edge in one of the next few matchings. This implies that the matchings $n_1, n_2, \ldots$ can be selected strategically so the processes will collide. More formally a the set of *good pairs* is defined below.

Definition 1 (Czumaj, Kutylowski). *Without loss of generality, assume X_0 and Y_0 differ by a (x, y) transposition and let $GP_0 = \{(x, y)\}$. For each $(x, y) \in GP_{t-1}$:*

1. *If neither x or y is part of the matching M_t then $(x, y) \in GP_t$.*

2. *If* $(x, w) \in M_t$ *and* y *is not part of* M_t *then* $(w, y) \in GP_t$.
3. *If* $(y, w) \in M_t$ *and* x *is not part of* M_t *then* $(w, x) \in GP_t$.
4. *If* (x, w), $(y, z) \in M_t$ *then if neither* w *or* z *are part of pairs in* GP_t *then* $(w, z) \in GP_t$ *and* $(x, y) \in GP_t$.
5. *Otherwise,* $(w, z) \in GP_t$.

In order to prove our theorem we need to show first that the after t_1 steps of the matching exchange process, there are a linear number of good pairs and then that after a t_2 additional steps we will select one of the good pairs as an edge in our matching. These two requirements are formalized in the two lemmas which we prove below. Although we provide more general bounds in terms of p_1 and p_2, the proof of our lemmas below relies heavily on techniques introduced by Miracle and Yilek [17]. Again for completeness we include the full analysis with our new parameters but the techniques remain the same as in [17]. Combining the lemmas with Czumaj and Kutylowski's coupling [7] and using the delayed path coupling theorem (Theorem 3) proves Theorem 1. More details can be found in [7] and [17].

Lemma 1. *Let* $|GP_t|$ *be the number of good pairs at step* t, *and* $t_1 = \max(40\ln(2n^2), 10\ln(n/9)/\ln(1 + p_1p_2(7/36)((7/9)n^2 - n)))$ *then*

$$\Pr[\,|GP_{t_1}| < n/9\,] \leq .5n^{-2}.$$

Proof. Initially (at step $t = 1$) there is one good pair namely, (x, y). At any step t, an existing good pair (x, y) contributes two good pairs to GP_{t+1} if both x and y are mapped to points that are not currently good pairs (see part 4 of Definition 1). We will start by bounding the probability that an existing good pair creates two good pairs at any step in a matching exchange process (i.e., part 4 is selected) in terms of p_1 and p_2. We begin by assuming that there are less than $n/9$ good pairs. If at any step t in the process there are more (i.e., $|GP_t| \geq n/9$) than we are done. Since each good pair contains two points, this implies there are at most $2n/9$ points in good pairs and at least $7n/9$ points not in good pairs. Recall that p_1 is the probability that a particular pair (x, y) is in the matching. Here we require not only that (x, y) is in the matching but that the associated bit-flip is true and the points are exchanged. Since there are least $7n/9$ points not in good pairs, the probability of a point x being match to a point not already in a good pair is $(p_1/2) * (7n/9))$. Note that the Recall that p_2 is the probability that conditioned on a first pair (x, y) being part of a matching, a second pair (w, z) is also in the matching. Given that x is matched to a point that is not a good pair there are at least $7n/9 - 1$ points remaining that are not in good pairs (i.e., not in GP_t) and thus the probability of y also being matched to a point that is not a good pair is at least $(p_2/2)(7n/9-1)$. Let p_4 be the probability that a particular good pair causes part 4 of the good pair definition to be selected (i.e., it results in two good pairs after the next step), then we have:

$$p_4 \geq (p_1 * p_2/4)(7n/9)(7n/9 - 1) = p_1p_2(7/36)((7/9)n^2 - n).$$

As shown in [17] if we let growth rate $G_t = (|GP_{t+1}| - |GP_t|)/|GP_t|$ then by linearity of expectations, $\mathbf{E}[G_t] = p_4$. Define an indicator random variable Z_t for when G_t exceeds one half it's expectation or $G_t \geq p_4/2$. Each time Z_t is one the number of good pairs increases at least by a factor of $1 + p_4/2$. This implies that if $\sum_{t=0}^{t_1} Z_t \geq \frac{\ln n/9}{\ln(1+p_4/2)}$ then the number of good pairs (i.e., $|GP_{t_1}|$) is at least $(1 + p_4/2)^{(\ln n/9)/\ln(1+p_4/2)} = n/9$, as desired. It is straightforward to show using Markov's inequality that $\Pr[Z_t = 0] \leq 4/5$ (see [17]). It is important to note that the Z_i's are not independent since the growth rate is more likely to be higher when there are fewer good pairs. In our analysis so far we are assuming there are always at most $n/9$ good pairs which holds throughout and thus this process is lower bounded by a process with independent variables $W_1, \ldots W_{t_1}$ where each variable W_i is 1 with probability $1/5$ and 0 with probability $4/5$ which will allow us to apply a Chernoff bound. We will use the following well-known bound $\Pr[W < \mathbf{E}[W]/2] < \exp(-\mathbf{E}[W]/8)$ with $W = \sum_{t=0}^{t_1} W_t$ and $t_1 = \max(40\ln(2n^2), 10\frac{\ln(n/9)}{\ln(1+p_4/2)})$. The selection of t_1 implies that $\mathbf{E}[W] \geq (1/5)40\ln(2n^2) = 8\ln(2n^2)$. Therefore,

$$\Pr[W < \mathbf{E}[W]/2] < \exp(-\mathbf{E}[W]/8) <= \exp(-8\ln(2n^2)/8) = .5n^{-2}.$$

And similarly, $\mathbf{E}[W] \geq (1/5)10\frac{\ln n/9}{\ln(1+p_4/2)} = 2\frac{\ln n/9}{\ln(1+p_4/2)}$. Together these prove Lemma 1,

$$\Pr\left[W < \frac{\ln n/9}{\ln(1+p_4/2)}\right] < \Pr[W < \mathbf{E}[W]/2] < .5n^{-2}.$$

□

The final component of our proof is to show that the matchings over the next t_2 steps will contain a good pair with probability at least $1 - .5n^{-2}$. As in [7,17], we say that a pair (x, y) is part of a *potential* matching if the process maps x to y regardless of the value of the bit-flip. Here, we are interested in the probability that a potential matching contains a good pair.

Lemma 2. *Let $t_2 = 72\ln(2n^2)/(p_1 n)$ then conditioned on $|GP_{t_1}| \geq n/9$, the probability that the next t_2 potential matchings contain no edges from GP_{t_1} is at most $.5n^{-2}$.*

Proof. First, consider a particular good pair (x, y). The probability that x is mapped to y at any step is p_1. There are at last $n/9$ good pairs and thus, by linearity of expectations, the expected number of good pairs in any potential matching is at least $p_1 n/9$. The potential matchings at each step in a matching exchange process are independent and thus we can apply same Chernoff bound above. Define the random variable E_t be the number of edges in the potential matching at time t that correspond to a good pair. Then using Chernoff, we have

$$\Pr\left[\sum_{t=t_1}^{t_1+t_2} E_t < 4\ln(2n^2)\right] < \exp(-8\ln(2n^2)/8) = .5n^{-2}$$

which directly implies that $\Pr\left[\sum_{t=t_1}^{t_1+t_2} E_t < 1\right] < .5n^{-2}$. □

References

1. Bellare, M., Hoang, V.T., Tessaro, S.: Message-recovery attacks on feistel-based format preserving encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16. pp. 444–455. ACM Press (Oct 2016)
2. Bellare, M., Ristenpart, T., Rogaway, P., Stegers, T.: Format-preserving encryption. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 295–312. Springer, Heidelberg (Aug 2009)
3. Bellare, M., Rogaway, P., Spies, T.: The FFX mode of operation for format-preserving encryption. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec.pdf` (February 2010)
4. Black, J., Rogaway, P.: Ciphers with arbitrary finite domains. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 114–130. Springer, Heidelberg (Feb 2002)
5. Brier, E., Peyrin, T., Stern, J.: BPS: A format-preserving encryption proposal. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/bps/bps-spec.pdf`
6. Brightwell, M., Smith, H.: Using datatype-preserving encryption to enhance data warehouse security. In: National Information Systems Security Conference (NISSC) (1997)
7. Czumaj, A., Kutylowski, M.: Delayed path coupling and generating random permutations. Random Structures & Algorithms 17, 238–259 (2000)
8. Durak, F.B., Vaudenay, S.: Break the FF3 format-preserving encryption standard over small domains. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part II. LNCS, vol. 10402, pp. 679–707. Springer (2017)
9. Dworkin, M.: Recommendation for block cipher modes of operation: Methods for format preserving-encryption. NIST Special Publication 800-38G, `http://dx.doi.org/10.6028/NIST.SP.800-38G` (2016)
10. Dyer, K.P., Coull, S.E., Ristenpart, T., Shrimpton, T.: Protocol misidentification made easy with format-transforming encryption. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 13. pp. 61–72. ACM Press (Nov 2013)
11. Grubbs, P., Ristenpart, T., Yarom, Y.: Modifying an enciphering scheme after deployment. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 499–527. Springer, Heidelberg (May 2017)
12. Hoang, V.T., Morris, B., Rogaway, P.: An enciphering scheme based on a card shuffle. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 1–13. Springer, Heidelberg (Aug 2012)
13. Levin, D.A., Peres, Y., Wilmer, E.L.: Markov chains and mixing times. American Mathematical Society (2006)
14. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. Journal of Cryptology 24(3), 588–613 (Jul 2011)
15. Luchaup, D., Dyer, K.P., Jha, S., Ristenpart, T., Shrimpton, T.: Libfte: A toolkit for constructing practical, format-abiding encryption schemes. In: Proceedings of the 23rd USENIX Security Symposium. pp. 877–891 (2014)
16. Luchaup, D., Shrimpton, T., Ristenpart, T., Jha, S.: Formatted encryption beyond regular languages. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 14. pp. 1292–1303. ACM Press (Nov 2014)

17. Miracle, S., Yilek, S.: Reverse cycle walking and its applications. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 679–700. Springer, Heidelberg (Dec 2016)
18. Morris, B., Rogaway, P.: Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 311–326. Springer, Heidelberg (May 2014)
19. Morris, B., Rogaway, P., Stegers, T.: How to encipher messages on a small domain. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 286–302. Springer, Heidelberg (Aug 2009)
20. Naor, M., Reingold, O.: Constructing pseudo-random permutations with a prescribed structure. Journal of Cryptology 15(2), 97–102 (2002)
21. Ristenpart, T., Yilek, S.: The mix-and-cut shuffle: Small-domain encryption secure against N queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 392–409. Springer, Heidelberg (Aug 2013)
22. Sinclair, A.: Algorithms for random generation and counting. Progress in theoretical computer science, Birkhäuser (1993)
23. Spies, T.: Format-preserving encryption. Unpublished whitepaper, available from `https://www.voltage.com/wp-content/uploads/Voltage-Security-WhitePaper-Format-Preserving-Encryption.pdf` (2008)