FastPRP: Fast Pseudo-Random Permutations for Small Domains

Emil Stefanov
UC Berkeley
emil@cs.berkeley.edu

Elaine Shi
UC Berkeley
elaines@cs.berkeley.edu

ABSTRACT

We propose a novel *small-domain pseudo-random permutation*, also referred to as a *small-domain cipher* or *small-domain (deterministic) encryption*. We prove that our construction achieves "strong security", i.e., is indistinguishable from a random permutation even when an adversary has observed all possible input-output pairs. More importantly, our construction is **1,000 to 8,000** times faster in most realistic scenarios, in comparison with the best known construction (also achieving strong security). Our implementation leverages the extended instruction sets of modern processors; and we also introduce a smart caching strategy to freely tune the tradeoff between time and space.

1. INTRODUCTION

Pseudo-random permutations (PRPs), also referred to as block ciphers, are at the foundation of modern cryptography. This paper investigates the problem of constructing pseudo-random permutations over a *small domain*.

Applications of small-domain PRPs. First proposed and studied by Black and Rogaway [7], small-domain PRPs are useful in a variety of application scenarios. For example, they are used in cryptographic constructions (e.g., Oblivious RAMs [9, 18]) for randomly reordering (permuting) a list of items. They can be used to generate pseudo-random unique tokens (e.g., product serial numbers) in a specific format. They can also be used to encrypt data in a small domain, such as encrypting a 9-digit social security number into another 9-digit number. Because of this, a small-domain PRP is also commonly referred to as a *small-domain cipher* or *format-preserving encryption* (FPE). FPE has been a useful tool in encrypting financial and personal identification information, and transparently encrypting information in legacy databases.

Challenges and requirements. The construction of small-domain PRPs presents unique technical challenges. First, note that applying a large-domain PRP such as a 128-bit AES and then projecting back into the domain $\mathcal{D}$ would destroy permutivity. In addition, there are several other challenges as suggested below.

- *Arbitrary domain size.* In this paper, we wish to construct small-domain PRPs that support arbitrary domain sizes, as opposed to fixed domain size, such as 16-bit or 32-bit.
- *Non-standard security requirement.* Standard large-domain PRPs over $\{0,1\}^\ell$ typically achieve security against an adversary who can issue up to $q \ll 2^\ell$ queries. With small-domain PRPs, the adversary might be able to exhaust all possible plaintext-ciphertext pairs. Our goal is to design a small-domain PRP that can withstand up to N queries from the adversary[1]. In other words, we require the small-domain PRP to be indisitinguishable from a random permutation, even to an adversary who has observed all N possible plaintext-ciphertext pairs.
- *Efficient point-wise evaluation with small time and space.* Another requirement is that the small-domain PRP should allow efficient point-wise evaluation. For example, one should be able to evaluate the outcome of the PRP over any input x in time and space *sublinear in* N.

1.1 Our Results and Contributions

As shown in Table 1, existing small-domain PRP schemes are either entirely based on empirical (but not provable) security [3, 4, 16] or fall short of security and only have provable security for a small number of adversary queries [7, 13] ($q \ll N$). Recently, Granboulan and Pornin [10] propose a novel small-domain PRP construction that can withstand N queries, and has provable security. While their scheme achieves non-trivial asymptotics, namely $O((\log N)^3)$ time and $O(\log N)$ space per invocation, it is generally considered to be impractical [6] due to the need to sample from the hypergeometric distribution, which is a heavy-weight operation.

We take an unconventional approach to solve this problem. We note that asymptotic performance is not the best metric to optimize for small domain problems. Instead of

[1]Note that in some earlier works, this same notion of security was also referred to as "withstanding $N-2$ queries", in the sense that the adversary cannot predict the remaining two outputs with probability more than $\frac{1}{2}$, even after observing $N-2$ input-output pairs.

Scheme	Arbitrary domain size*	Secure for $q \ll N$ queries**	Secure for $q = N$ queries**
[3, 4, 16, 17]	No	No	No
[6, 7, 11, 13, 15]	Yes	Yes	No
[10]	Yes	Yes	Yes
This paper	**Yes**	**Yes**	**Yes** (1,000∼8,000 times faster than [10])

*: Note that cycle-walking can be used in general to achieve arbitrary domain size, but is usually quite costly.
**: By "secure", we mean provably (rather than empirically) indistinguishable from a random permutation even when the adversary has seen any q possible input-output pairs.

Table 1: Comparison with related work

proposing a construction that is asymptotically better, we propose one that is asymptotically worse – $O(\sqrt{N}\log N)$ in both time and space – but is **1,000** to **8,000** times faster than the best existing scheme [10] for the most common use cases (domains of sizes up to $N = 2^{32}$ items). Our construction is currently by far the most practical construction for most realistic scenarios. In addition, our construction can withstand up to N queries by the adversary, and its security formally reduces to the security of the underlying AES function, which is used to generate the pseudo-random source needed by our construction.

1.2 Technical Highlights

No costly sampling from hypergeometric distributions. One approach taken by Granboulan and Pornin [10] to construct a random permutation is to recursively partition (at random) a set of N items into two equally sized sets, until each set has size 1. The task of partitioning a set of N items into two equally sized sets can essentially be transformed into the task of generating N random bits of which exactly $\frac{N}{2}$ are 0 and $\frac{N}{2}$ are 1. Generating this bitstring turns out to be a difficult problem because it seems to require sampling from the hypergeometric distribution with very high accuracy in order to preserve security.

We observe that it is possible to avoid the costly sampling by relaxing the requirements and allowing there to be slightly more ones or zeros in the bitstring. In our construction, each bit is randomly and independently generated and so the total number of ones or zeros is a distribution centered on $\frac{N}{2}$.

Leveraging modern processor features. Our implementation is highly optimized and takes advantage of modern processors' extended instructions, including `AES` instructions and the `POPCNT` instruction. These advanced x86-64 instructions are widely available on most modern server, desktop, and laptop processors.

Even though existing algorithms can also be modified to use those instructions, they do not benefit nearly as much. In fact, we modified the code of [10] to use those hardware instructions and our algorithm is still thousands of times faster.

Tunable time-space tradeoff. We propose a smart caching strategy (Section 4) to cache and reuse intermediate computation results – specifically, counters, as described in Section 4 – allowing us to evaluate the permutation in $O(\sqrt{N}\log N)$ time and space.

Our scheme is dynamically adjustable to the amount of cache available. Specifically, at run-time, if the amount of available memory changes, our scheme can dynamically adjust and maximally leverage the available memory to minimize the computation time.

1.3 Related Work

The problem of construct a small-domain PRP or small-domain cipher was initially studied by Black and Rogaway [7].

One class of approaches is balanced or unbalanced Feistel-based constructions [6,11,13,15]. However, these approaches do not have proven security against up to N queries when applied to small domains. Let $N = 2^n$ denote the size of the domain. For balanced cipher, Luby and Racoff [11] showed that 4 rounds can provide security to nearly $2^{n/4}$ queries. Maurer and Pietrzak [12] show that r rounds of balanced Feistel could withstand about $2^{n/2-1/r}$ queries. Patarin [14, 15] shows that 6 rounds is enough to withstand about $2^{n/2}$ queries. Morris *et al.* analyze the security of the Thorp shuffle [13], which is a maximally unbalanced Feistel network. They show that the Thorp shuffle with $O(r)$ rounds is resilient up to $2^{n(1-1/r)}$ queries.

Another class of approaches is *de novo* constructions [3, 4, 16, 17]. These constructions have empirical, but not provable security. Some of them only work for fixed length block ciphers. Moreover, due to lack of provable security, significant attacks were found with the TEA construction, leading to the hacking of Microsoft XBox console.

Small-domain ciphers are a special case of format-preserving encryption, which was informally described by Brightwell and Smith [8], named by Spies, and recently received a more systematic treatment by Bellare and Ristenpart [5].

Table 1 compares our work against related work in this space.

2. PROBLEM DEFINITION

We first formally define pseudo-random permutations and their security requirement.

DEFINITION 1. *Let $\mathcal{D} := \{0, 1, \ldots, N-1\}$ denote a finite domain of elements. Let $\mathcal{K} := \{0,1\}^k$ denote the key space. Let $P : \mathcal{K} \times \mathcal{D} \to \mathcal{D}$ be a permutation that maps $\mathcal{D}$ onto itself. P is a (t, q, ϵ)-pseudo-random permutation, if*

- *For any $K \in \mathcal{K}$, $\mathsf{PRP}(K, \cdot)$ is a one-to-one function from $\mathcal{D}$ to $\mathcal{D}$, and can be evaluated in polynomial time.*
- *For any t-time algorithm $\mathcal{A}$ making at least q oracle queries,*

$$\Pr_{k \xleftarrow{\$} \mathcal{K}}[\mathcal{A}^{\mathsf{PRP}(K,\cdot),\mathsf{PRP}^{-1}(K,\cdot)} = 1] - \Pr_{p \xleftarrow{\$} \mathcal{P}}[\mathcal{A}^{p,p^{-1}} = 1] \leq \epsilon$$

where $\mathcal{P}$ denotes the family of all permutations for $\mathcal{D}$.

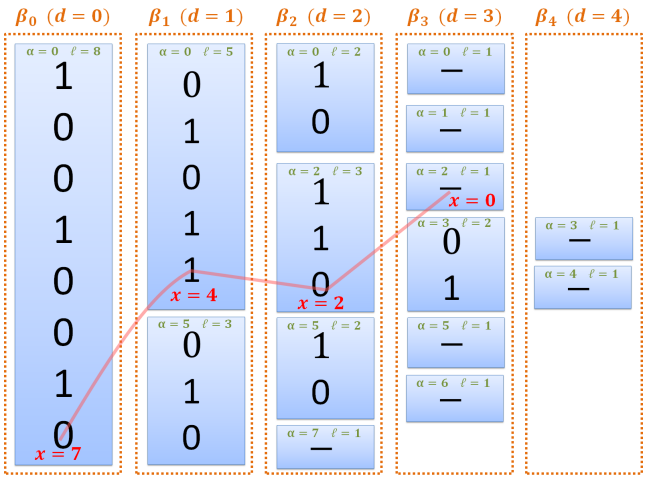

Figure 1: Example of the permutation algorithm.

In particular, in this paper, we would like to construct pseudo-random functions a small domain $\mathcal{D} := \{0, 1, \ldots, N-1\}$ that are **secure against up to $q = N$ queries**.

3. ALGORITHM

3.1 Intuition

Suppose that we would like to randomly permute N elements from a domain $\mathcal{D} := \{0, 1, \ldots, N-1\}$. The permutation can be done in the following manner:

- Choose a random bit $\beta[i] \in_R \{0, 1\}$ for every $i \in \mathcal{D}$.
- Let $S_0 := \{i \mid i \in \mathcal{D} \text{ and } \beta[i] = 0\}$ denote the set of elements whose $\beta[i]$ values are 0, let $S_1 := \{i \mid i \in \mathcal{D} \text{ and } \beta[i] = 1\}$ denote the set of elements whose $\beta[i]$ values are 1.

  Place the set S_0 ahead of S_1 in the final permutation.
- Recurse and permute both sets S_0 and S_1.

The above randomized algorithm generates a random permutation over the domain $\mathcal{D} := \{0, 1, \ldots, N-1\}$. For the formal proof of this, please refer to Section 3.5. To construct a pseudo-random permutation, one can simply use a pseudo-random source in place of the random bits fed to the algorithm. For instance, we can use $\mathsf{AES}_K(\cdot)$ to generate the pseudo-random bits of β to obtain a keyed pseudo-random permutation where the secret key is K.

Example. Figure 1 illustrates the informal algorithm described above, using a small domain example, where $\mathcal{D} := \{0, 1, \ldots, 7\}$. To find the outcome of the pseudo-random permutation on the input $x = 7$, one first assigns a random bit to each of the elements in $\mathcal{D} := \{0, 1, \ldots 7\}$. The vector of these random bits form the bitstring β_0. Since $x = 7$ gets assigned the random bit 0, it will be placed in the top partition. Now this process is recursively applied to the top partition, where the element was mapped to, until a partition of size 1 is reached. At that point, this pseudo-random permutation algorithm determines that the final location of input 7 is 2.

3.2 An Alternative View of the Algorithm

The algorithm (informally) described in Figure 1 is equivalent to the following process.

- First, for each input $i \in \{0, 1, \ldots, N-1\}$, assign a random $O(\log N)$-bit number ρ_i to element i.
- Next, sort all i's based on their ρ_i values. With high probability, all N elements will be assigned a unique ρ_i, which determines a unique ordering of all N elements.

It is not hard to see that the above-mentioned procedure outputs a random permutation of all elements $i \in \{0, 1, \ldots, N-1\}$.

Moreover, it is not hard to see that Figure 1 effectively implements this above procedure, where the sorting is accomplished through a *radix-sort* process. In particular, in the first step of the recursion, the inputs are sorted based on the first bit of each ρ_i. We then recurse on this, and in depth d of the recursion, the d-th bit is being sorted.

Based on this alternative view of the algorithm, it is not hard to see that if the bit-strings $\beta_0, \beta_1, \ldots$ are generated at random, then the above process yields a random permutation. Similarly, if the bit-strings $\beta_0, \beta_1, \ldots$ are generated from a pseudo-random sequence, the above process yields a pseudo-random permutation. The formal security statements and proofs are presented in Section 3.5.

3.3 Notations

We first introduce some notations before formally presenting the detailed construction.

Pseudo-random bitstrings β_d. The bitstrings β_d are indexable arrays of pseudo-random independent bits. Each bitstring is N bits long. The value of bit i of bitstring β_d is denoted as $\beta_d[i]$.

Bit counters C_0 and C_1. The number of zeros in $R = \{\beta_d[\alpha], \beta_d[\alpha+1], \ldots, \beta_d[\alpha+x]\}$ is denoted as $\mathsf{C}_0(\beta_d, \alpha, x)$. Similarly, the number of ones is denoted as $\mathsf{C}_1(\beta_d, \alpha, x)$. The set R is called the input range of C_0 and C_1.

Bit locators C_0^{-1} and C_1^{-1}. The index of the k'th zero bit in $R = \{\beta_d[\alpha], \beta_d[\alpha+1], \ldots\}$ is denoted as $\mathsf{C}_0^{-1}(\beta_d, \alpha, k)$. Similarly, the index of the k-th one bit is denoted as $\mathsf{C}_1^{-1}(\beta_d, \alpha, k)$. The set R is called the input range of C_0^{-1} and C_1^{-1}.

3.4 Detailed Construction

Recall that we wish to construct a small-domain pseudo-random permutation, $\mathsf{PRP} : \mathcal{K} \times \mathcal{D} \to \mathcal{D}$, where $\mathcal{D} := \{0, 1, \ldots, N-1\}$ represents the domain, and $\mathcal{K} := \{0,1\}^k$ represents the key space.

Generation of pseudo-random source. The Permute and Unpermute algorithms below require pseudo-random bits as inputs. For notational convenience, we will divide these pseudo-random bits into bitstrings, denoted as $\{\beta_0, \beta_1, \ldots, \}$, where each $\beta_i \in \{0,1\}^N$ is a bitstring of length N.

These pseudo-random bits are generated with the key $K \in \mathcal{K}$ to the small-domain PRP, by applying the AES function:

$$S = \mathsf{AES}_K(0) || \mathsf{AES}_K(1) || \mathsf{AES}_K(2) || \ldots, \qquad (1)$$

In particular, β_0 will be the first N bits of S, β_1 will be the next N bits of S, and so on.

Permute. To compute $\mathsf{PRP}(K, x)$, where $K \in \mathcal{K}$, $x \in \mathcal{M}$, simply call the recursive function shown in Figure 2

```
Permute(x, α, ℓ, d):
 1: if ℓ = 1 then
 2:     return α
 3: end if
 4: if β_d[α + x] = 0 then
 5:     x' ← C_0(β_d, α, x)
 6:     return Permute(x', α, C_0(β_d, α, ℓ), d + 1)
 7: else
 8:     x' ← C_1(β_d, α, x)
 9:     return Permute(x', α + C_0(β_d, α, ℓ), C_1(β_d, α, ℓ), d + 1)
10: end if
```

Figure 2: Permutation algorithm (encryption).

```
Unpermute(y, α, ℓ, d):
 1: if ℓ = 1 then
 2:     return 0
 3: end if
 4: if y < C_0(α, ℓ, d) then
 5:     y' ← y
 6:     x' ← Unpermute (y', α, C_0 (β_d, α, ℓ) , d + 1)
 7:     return C_0^{-1} (β_d, α, x' + 1)
 8: else
 9:     y' ← y − C_0 (β_d, α, ℓ)
10:     x' ← Unpermute (y', α + C_0 (β_d, α, ℓ) , C_1 (β_d, α, ℓ) , d + 1)
11:     return C_1^{-1} (β_d, α, x' + 1)
12: end if
```

Figure 3: Inverse permutation algorithm (decryption).

with $\mathsf{Permute}(x, 0, N, 0)$. Specifically, the pseudo-random bits $\beta_0, \beta_1, \ldots$ required in this algorithm are generated as in Equation 1 above.

Figure 1 is an example walk-through of this algorithm for a small domain $\mathcal{D} := \{0, 1, \ldots, 7\}$. In Section 7, we show that the algorithm will terminate within depth d of $O(\log N)$ with high probability.

Unpermute. To compute $\mathsf{PRP}^{-1}(K, y)$, where $K \in \mathcal{K}$, $y \in \mathcal{M}$, simply call the recursive function shown in Figure 3 with $\mathsf{Unpermute}(y, 0, N, 0)$. Specifically, the pseudo-random bits $\beta_0, \beta_1, \ldots$ required in this algorithm are generated as in Equation 1 above.

Since Unpermute is the inverse function of Permute, and shares the same recursion tree as Permute, its depth is also bounded by $O(\log N)$ with high probability (Section 7).

3.5 Security Analysis

The alternative view on the algorithm described in Section 3.2 immediately gives us the following theorem:

THEOREM 1 (RANDOM PERMUTATION). *Assuming the bit vectors $\beta_0, \beta_1, \ldots$, are chosen at random, where each bit represents the outcome of an indenpendent random coin flip. The algorithm described in Figure 2 yields a perfectly random permutation over elements in $\mathcal{D} = \{0, 1, \ldots, N-1\}$.*

PROOF. Immediately follows due to the alternative view of the algorithm described in this section. □

Theorem 1 assumes that the bit-strings $\beta_0, \beta_1, \ldots$, are chosen at random. Instead, if these bit-strings are generated pseudo-randomly from AES, one can show that the resulting small-domain PRP is "at least as secure as" AES. To prove this, one has to ensure that the permutation algorithm has bounded depth, such that the bit-strings $\beta_0, \beta_1, \ldots$, can be obtained from a bounded number of AES invocations. In particular, k invocations of AES would lead to a multiplicative factor of k in the advantage of the adversary.

COROLLARY 1. *Assume the bit vectors $\beta_0, \beta_1, \ldots$ are obtained from a pseudo-random sequence, generated by applying AES as in Equation 1. Suppose the underlying AES is a (t, q, ϵ)-pseudo-random permutation, where $q \geq 5 \log N$. Then, the algorithm described in Figure 2 yields a $(t, N, \epsilon + \mathsf{negl}(q))$-pseudo-random permutation.*

PROOF. Suppose that there exists an adversary $\mathcal{A}$ who can break the small-domain permutation PRP. We now leverage this adversary $\mathcal{A}$ to construct a simulator $\mathcal{B}$ which can break AES.

Basically, $\mathcal{A}$ submits a sequence of queries to $\mathcal{B}$, and $\mathcal{B}$ will simulate the small-domain PRP function, and evaluate the outcomes of the PRP function for $\mathcal{A}$. $\mathcal{B}$ obtains its bit-strings $\beta_0, \beta_1, \ldots$ from an oracle, which either is the AES function or a pure random source. With probability $1 - \mathsf{negl}(q)$ (due to a chernoff bound argument similar to the proof of Theorem 2), $\mathcal{B}$ only has to make q or fewer queries to the oracle. If, however, this fails, $\mathcal{B}$ simply aborts.

Then, $\mathcal{B}$ basically outputs whatever $\mathcal{A}$ outputs. It is not hard to see that if $\mathcal{A}$ succeeds in distinguishing PRP from a random permutation, then $\mathcal{B}$ would be able to distinguish whether the oracle is a pseudo-random source generated

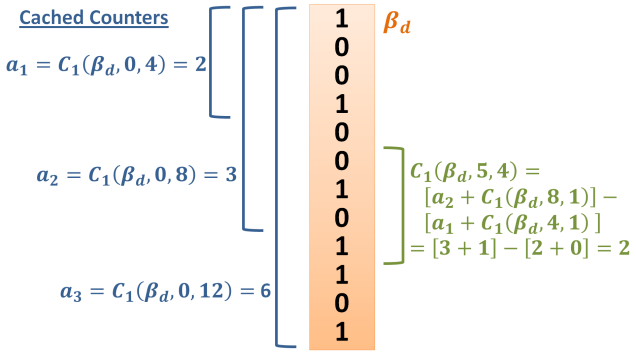

Figure 4: Using cached values of $\mathsf{C}_1(\cdot)$ to efficiently compute arbitrary values of $\mathsf{C}_1(\cdot)$. In this example, $\mathsf{C}_1(\beta_d, 5, 4)$ is computed using the cached values of $\mathsf{C}_1(\beta_d, 0, 4)$ and $\mathsf{C}_1(\beta_d, 0, 8)$ and a small amount of scanning to calculate the values of $\mathsf{C}_1(\beta_d, 4, 1)$ and $\mathsf{C}_1(\beta_d, 8, 1)$.

from AES or a pure random source, thereby breaking the security of AES. □

4. CACHING COUNTERS

The scheme will be very slow if we implement the Permute and Unpermute algorithms exactly as shown in Figures 2 and 3. The reason is that a naive implementation of the C_0, C_1, C_0^{-1}, and C_1^{-1} functions cause a linear scan of large ranges of β_d. The total length of the scanned regions is $O(N)$, so a naive implementation of Permute and Unpermute would result in $O(N)$ running time, which is too slow in practice.

We reduce the time complexity of Permute and Unpermute to $O(\sqrt{N}\log N)$ with $O(\sqrt{N}\log N)$ amount of space, by *caching* a small number of intermediate counters of β with a granularity s (called the cache stride).

DEFINITION 2 (CACHE STRIDE s). *The cache stride determines the interval at which the values of $\mathsf{C}_1(\cdot)$ are cached. With a cache stride s, the values of $\mathsf{C}_1(\beta_d, 0, s \cdot i)$ are cached for $d = 0, 1, \ldots, O(\log N)$ and $i = 1, 2, \ldots, N/s$.*

Generating the cache takes $O(N \log N)$ time, but is performed only once at initialization. In our experiments in Section 6.1, we show that it can be done reasonably fast (less than 10^{-6} seconds for $N < 2^{15}$ to 2.3 seconds for $N = 2^{31}$). After this one-time initialization, we can reuse the cache to evaluate multiple "ciphertexts" with the same key.

Computing C_1 with a cache. Figure 4 illustrates the the cached counters for β_d with $N = 12$ and $s = 4$. The values $a_1 = \mathsf{C}_1(\beta_d, 0, 4 \cdot 1)$, $a_2 = \mathsf{C}_1(\beta_d, 0, 4 \cdot 2)$, and $a_3 = \mathsf{C}_1(\beta_d, 0, 4 \cdot 3)$ are stored in a lookup table. If the algorithm needs to compute $\mathsf{C}_1(\beta_d, 5, 4)$, it can calculate it as $[a_2 + \mathsf{C}_1(\beta_d, 8, 1)] - [a_1 + \mathsf{C}_1(\beta_d, 4, 1)]$.

This saves computation time because the algorithm now only needs to scan a range of size 2 instead of size 4. Of course, with larger and more realistic values of N and s, the savings are much greater. In fact, any C_1 function will have to scan at most $2s$ bits, even if the counting range is much larger (e.g., $O(N)$).

Although the algorithm works for any values of s, for our experiments, we have chosen $s = 2\sqrt{N}$ because it provides a balanced trade-off between computation time and key size. As a result, the C_1 function will never need to scan more than $2s = 4\sqrt{N}$ bits. With this cache stride, we can guarantee that any C_1 function will have to scan at most $4\sqrt{N}$ bits, even if the range covers $O(N)$ bits.

Computing C_0, C_0^{-1}, and C_1^{-1} with a cache. The functions C_0, C_0^{-1}, and C_1^{-1} can all be calculated efficiently from the cached values of C_1.

For C_0, we can simply count the ones in the input range with the cache-optimized C_1 function and then subtract the count from the size of the range as follows:

$$\mathsf{C}_0(\beta_d, \alpha, x) = x - \mathsf{C}_1(\beta_d, \alpha, x)$$

Computing $\mathsf{C}_1^{-1}(\beta_d, a, k)$ is slightly different since it requires a binary search of the cached values of C_1. First, we set

$$k' = \mathsf{C}_1(\beta_d, 0, \alpha) + k$$

by using the cache-optimized C_1 function. We then binary search the cache over $i = 1, 2, \ldots, N/s$ for i such that

$$\mathsf{C}_1(\beta_d, 0, s \cdot i) < k' \leq \mathsf{C}_1(\beta_d, 0, s \cdot (i+1))$$

This allows us to compute C_1^{-1} as follows:

$$\mathsf{C}_1^{-1}(\beta_d, a, k) = \mathsf{C}_1^{-1}(\beta_d, s \cdot i, k' - \mathsf{C}_1(\beta_d, 0, s \cdot i))$$

Similarly, C_0^{-1} can also be computed with a binary search.

Levels to cache. Note that in Definition 2, the values of $\mathsf{C}_1(\beta_d, \cdot)$ are cached for $d = 0, 1, \ldots, O(\log N)$. In practice, it is not necessary to cache levels beyond about $d = \log_2(N/s)$ because the expected size of a input ranges for C_1 at that depth is less than s. For larger values of d, C_1 can be implemented as a linear scan. In our implementation we stopped caching after depth $d = \log_2(N/s)$.

4.1 Bidirectional Scanning

Caching C_1 counter values significantly reduces the need for linearly scanning bits of β_d. However, it does not completely eliminate scanning. In order to compute values of C_1 for arbitrary inputs, we need to scan s bits on average and at most $2s$ bits with forward scanning (i.e., in order of increasing bit indexes of β_d).

As illustrated in Figure 5, we can further reduce the amount of scanning in half if we extend the algorithm to start from a cached counter boundary and scan backwards until it reaches an edge of the input range. Backward scanning should be performed when the forward scanning length is more than $s/2$ (half of the caching stride) and the result should be subtracted from the cached counter value. This optimization makes it so that the average bits scanned per invocation of the C_1 (and hence C_0) functions is $s/2$ and the maximum is s. The same trick can be used for C_0^{-1} and C_1^{-1}.

4.2 Counter Alignment

The location of counter boundaries can be slightly shifted to align with the edges of partitions. For example, in Figure 1, without counter alignment and a stride of $s = 4$, we would store counters for $\mathsf{C}_1(\beta_1, 0, 4)$ and $\mathsf{C}_1(\beta_1, 4, 4)$. If we align the counters, then we can instead store the counters for $\mathsf{C}_1(\beta_1, 0, 5)$ and $\mathsf{C}_1(\beta_1, 5, 3)$.

Aligning counters helps boost performance. As can be seen in Figures 2 and 3, the algorithm counts the number of ones in the current partition at each level of recursion.

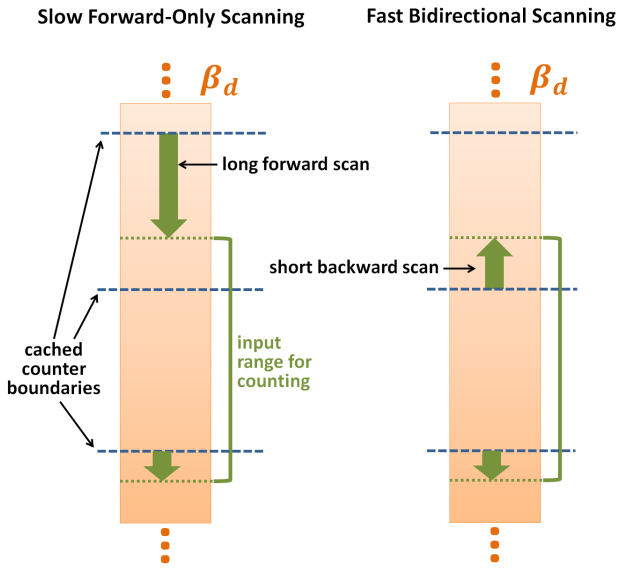

Figure 5: Performing bidirectional scanning instead of forward-only scanning leads to a 2X speedup of the algorithm.

Therefore aligning the counters with the partition boundaries allows the algorithm to count the number of ones bits in a partition by using the cache only and completely avoids linearly scanning the bitstrings in these cases. Moving the cache boundaries comes at a cost of making other C_1 operations slightly less efficient, but because less efficient operations occur with lower frequency, counter alignment actually improves the performance of the overall algorithm.

5. ENHANCEMENTS

5.1 Optimization via Assembly Instructions

The most time-consuming operation in FastPRP is the linear scanning of the bitstrings β_d, which happens as a result of calling the C_0, C_1, C_0^{-1}, or C_1^{-1} function. In Section 4, we explain how the amount of scanning can be significantly reduced by caching a small number of counters. However, the scanning cannot be completely eliminated, and the number of bits scanned per invocation of C_0, C_1, C_0^{-1}, or C_1^{-1} is $\Theta(\sqrt{N}\log N)$.

We observe that this scanning operation can be performed extremely efficiently with the x86 aesenc, aesenclast, and popcnt assembly instructions. These instructions are available on most modern server, desktop, and laptop processors. On other processors, such as those for mobile phones, the scanning can be done in software without hardware acceleration.

AES instructions. Modern x86 processors offer the aesenc and aesenclast instructions which perform one round of an AES encryption as a single instruction. Encrypting a single block takes several rounds (e.g., 10 rounds for 128-bit AES and 14 rounds for 256-bit AES). In our implementation we use 128-bit AES, but this can easily be adjusted (e.g, by adding 4 extra rounds to make it 256-bit).

```
; Input AES round keys:     xmm0-xmm10
; Input block ID:           xmm15
; Output:                   rax (# ones in xmm15)

; xmm15 = AesEncrypt(xmm15)
pxor xmm15, xmm0 ; Whitening step (AES Round 0)
aesenc xmm15, xmm1 ; AES Round 1
aesenc xmm15, xmm2 ; AES Round 2
aesenc xmm15, xmm3 ; AES Round 3
aesenc xmm15, xmm4 ; AES Round 4
aesenc xmm15, xmm5 ; AES Round 5
aesenc xmm15, xmm6 ; AES Round 6
aesenc xmm15, xmm7 ; AES Round 7
aesenc xmm15, xmm8 ; AES Round 8
aesenc xmm15, xmm9 ; AES Round 9
aesenclast xmm15, xmm10 ; AES Round 10

; rax = Count ones bits in xmm15
movq r8, xmm15
psrldq xmm15, 8
movq r9, xmm15
popcnt rbx, r8
add rax, rbx
popcnt rcx, r9
add rax, rcx
```

Figure 6: Highly efficient x86-64 assembly instructions for counting the ones bits in a single AES block.

POPCNT instruction. The popcnt (population count) instruction conveniently allows us to count the number of ones bits in a 64-bit register. Without this instruction, we would need to use less efficient methods such as those in [1].

Example. Suppose that we need to scan β_0 to count the number of ones bits in $S = \{\beta_0[0], \beta_0[1], ..., \beta_0[2559]\}$. Recall that bitsrings are generated by consecutive calls to $\mathsf{AES}_K(\cdot)$ with an incrementing index. Specifically, S is the following bitstring:

$$S = \mathsf{AES}_K(0)||\mathsf{AES}_K(1)||\mathsf{AES}_K(2)||\ldots||\mathsf{AES}_K(19)$$

The algorithm works as follows: Let $c = 0$. For $i = 0, ..., 19$, compute $\mathsf{AES}_K(i)$, count the number of ones bits in it, and add the count to c. Figure 6 shows assembly code executed for a single value of i. The index i is passed in as register xmm15. The first block of code uses the AES instructions aesenc and aesenclast to compute $\mathsf{AES}_K(i)$. The second block of code uses the popcnt instruction to count the number of bits in $\mathsf{AES}_K(i)$.

In cases where the bitstring partially covers an AES block, the ones bits in the first and last block of the bitstring can be counted by reading individual bits. The bulk of the AES blocks (in between the first and last block) can still be scanned using the efficient assembly code in Figure 6.

This highly optimized scanning process is used in the implementation of the C_1 function to scan bitsrings immediately before and after cached counter boundaries as described in Section 4. Similar assembly code can be used to perform the scanning for C_0, C_0^{-1}, and C_1^{-1}.

5.2 Cache Compression

The cache for FastPRP can vary from 365 bytes for $N =$

2^{11} to 893KB for $N = 2^{31}$. Typically, this is a very small amount of memory usage to get the performance and security that FastPRP offers. However, in some scenarios, the user may want to store many keys in memory or the amount of memory could be severely limited such as in embedded devices.

Recall that the cache consists of counters for regions of length s in the bitstrings. Each counter specifies the number of ones in a particular region. Since all of the bits are essentially independent random coin flips, we know that the counter value is a random variable with a binomial distribution. Hence the entropy of the counter for the number of ones bits in a region of size s is

$$\frac{\log_2(\pi e s) - 1}{2} + O\left(\frac{1}{s}\right)$$

We can use Huffman codes to compactly store the counter values. This means that, for example, for $N = 2^{31}$ and $s = 2^{16}$, we can use about 9 bits per counter on average even though the counters are 16-bit numbers, resulting in about a 57% compression ratio.

5.3 Incremental Caching

The cache size can always be increased or decreased at run time. For example, if the program is expecting to make lots of PRP invocations in the near future, it can temporarily increase the cache size for improved performance.

It is possible to store the cache in increments where each increment cuts the cache stride in half. For example suppose that $N = 16$. In increment I_1, we can cache

$$I_1 = \{\mathsf{C}_1(\beta_d, 0, 8), \mathsf{C}_1(\beta_d, 8, 8)\}$$

and in increment I_2, we can cache

$$I_2 = \{\mathsf{C}_1(\beta_d, 0, 4), \mathsf{C}_1(\beta_d, 8, 4)\}$$

Increment I_1 is itself a cache with stride $s = 8$. However, we can combine I_1 and I_2 to obtain a cache with stride $s = 4$. Note that this is possible because the remaining cache values $\mathsf{C}_1(\beta_d, 4, 4)$ and $\mathsf{C}_1(\beta_d, 12, 4)$ can be trivially computed from I_1 and I_2 without performing any scanning.

Increments can also be stored on disk as separate files and loaded on-demand when higher performance is desired.

6. EVALUATION

To evaluate our algorithm, we implemented our FastPRP Permute and Unpermute algorithms for arbitrary values of N. Our implementation uses an adjustable cache size as described in Section 4 and the assembly instruction optimizations described in Section 5.1. We chose a cache stride of $s = 2\sqrt{N}$ for our experiments because it offers a balanced tradeoff between the cache size and speed. We ran all of our experiments on the same 4 GHz Intel Core i7 CPU machine.

6.1 Performance

Permute/Unpermute. Figure 7 shows the speed of our algorithm's Permute and Unpermute operations for domains of size $N = 2^{11}$ to $N = 2^{31}$. Each data point is an average of 2^{17} operations with uniform random inputs. As one would expect, the performance of the algorithm decreases as N increases because the recursion depth increases and the cache stride $s = 2\sqrt{N}$ increases with N.

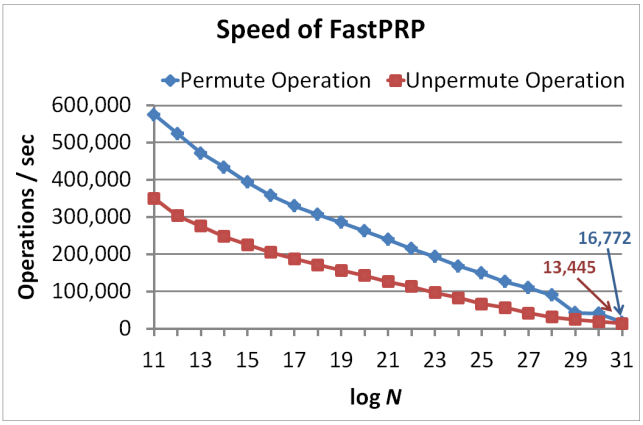

Figure 7: The performance of our construction with cache stride of $s = 2\sqrt{N}$.

$\log N$	11	15	21	25	31
Size	365 B	1.9 KB	20 KB	92 KB	893 KB
Time (s)	$< 10^{-6}$	$< 10^{-6}$	0.001	0.031	2.34

Table 2: FastPRP cache sizes and cache generation time.

The Unpermute operation is slower than the Permute operation because: (1) in our implementation we did not implement backward scanning for C_0^{-1} and C_1^{-1} functions invoked by Unpermute, and (2) the C_0^{-1} and C_1^{-1} functions do an additional binary search as explained in Section 4.

Cache generation. Generating the cached counters requires a scan of bitstrings β_d for $d = 0, 1, \ldots, \log_2(N/s)$. This might be concerning because for $N = 2^{31}$, this consists of scanning bitstrings of combined length of 2^35 bits. Our results in Table 2 show that the cache generation time is actually quite fast; even for $N = 2^{31}$, it takes about 2.34 seconds on a single core to generate the key. In comparison, it takes about 1.2 seconds to generate a 3072-bit RSA key on the same hardware using OpenSSL. According to NIST [2] a 3072-bit RSA key provides about the same level of security as the 128-bit AES key used by our implementation. It's important to note that the cache generation only needs to be done once and the cache can then be stored along with the key for future use.

6.2 Comparison to Previous Work

As explained in Section 1.3 and Table 1, the only previous work that achieves as strong security guarantees (withstanding N queries and provable security) as FastPRP is a construction by Granboulan and Pornin [10]. We contacted the authors and they kindly provided us with the implementation that they used in [10]. To achieve a fair comparison, we modified their code to use the same hardware AES instructions as FastPRP (described in Section 5.1).

For domains of size $N = 2^{11}$ to $N = 2^{31}$, we measured the amount of time it takes FastPRP to perform 2^{17} Permute and Unpermute operations with uniform random inputs. We timed the same operations for [10], but because that algorithm is much slower, we used 1000 measurements per data point.

As the results show in Figure 8, our construction is **1,000**

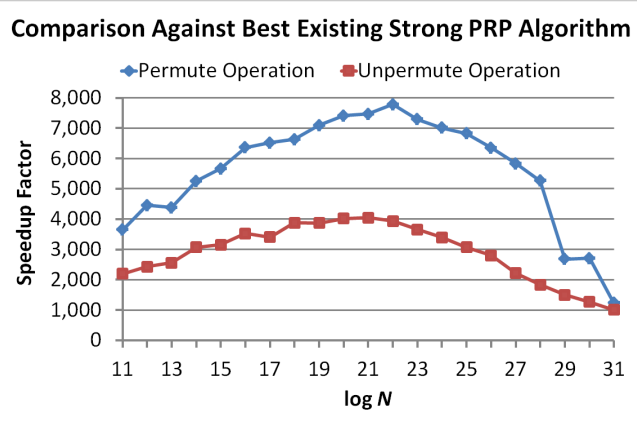

Figure 8: Speed comparison between our algorithm and the best previously known algorithm [10] with the same level of security. The "speedup" measured is how many times faster our construction is than the algorithm in [10]. We used cache stride of $s = 2\sqrt{N}$ for our algorithm.

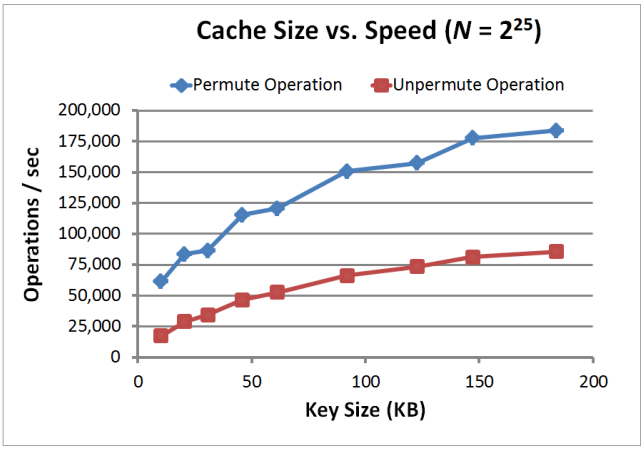

Figure 9: The tradeoff between cache size and speed of FastPRP.

to 8,000 times faster than the best existing construction with the same level of security. Because our algorithm has a higher *asymptotic* complexity than [10], as N keeps increasing eventually their construction will become faster. Unfortunately, we do not know the exact value of N because determining it will require a significant change to their code to handle values with more than 31 bits. However, the graph clearly shows that for $N \leq 2^{31}$, our construction is much faster in practice.

6.3 Cache Size vs. Speed

In Figure 9, we demonstrate the tradeoff between cache size and speed of our algorithm. As the cache stride s is decreased for a fixed value domain size ($N = 2^{25}$ in this graph), the cache size increases and so does the speed of the Permute and Unpermute operations. The reason behind this is that a smaller stide between the cached counters reduces the linear scanning work performed by the C_0, C_1, C_0^{-1}, C_1^{-1} functions.

7. ASYMPTOTIC ANALYSIS

As mentioned in Section 1, asymptotics is not the most important metric to optimize with small-domain problems. For completeness, in this section, we formally prove that with high probability, the depth of the permute algorithm is bounded by $O(\log N)$.

THEOREM 2. *With probability* $1-O(\frac{1}{N^2})$*, the depth of the* Permute *(or* Unpermute*) algorithm is bounded by* $O(\log N)$.

PROOF. First, we show that for each input $i \in \mathcal{D} := \{0, 1, \ldots, N-1\}$ to the pseudo-random permutation, the depth for element i is bounded by $O(\log N)$ with high probability. Next, we apply union bound over the set of all elements in $\mathcal{D}$.

Suppose an element $i \in \mathcal{D}$ is in some partition S in the k-th iteration. This element $i \in \mathcal{D}$ is called *lucky* in the k-th iteration, if this iteration divides S into two parts, where both parts contain at most $\lceil \frac{3}{4}|S| \rceil$ elements. Clearly, element i can participate in at most $\log_{\frac{4}{3}} N$ lucky rounds.

It is not hard to see that any round is a lucky round with probability at least $\frac{1}{2}$, for the following reason.

CLAIM 1. *Due to Chernoff bound, in a sequence of* M *coin flips, the probability that the number of ones is smaller than* $\frac{M}{4}$ *is at most* $\exp(-M/8)$.

Therefore, for rounds where $M \geq 8$, clearly the probabilty of a lucky round is at least $\frac{1}{2}$. For $M < 8$, it is not hard to verify that the probabilty of a lucky round is at least $\frac{1}{2}$ as well.

Again, due to Claim 1, in a sequence of $8 \log_{\frac{4}{3}} N$ rounds,

$$\Pr[\text{number of lucky rounds} < \log_{\frac{4}{3}} N]$$
$$\leq \exp(-8 \log_{\frac{4}{3}} N/8) \leq \frac{1}{N^3}$$

Given the above, and taking union bound over all N elements, we can prove Theorem 2.

For the Unpermute algorithm, since it is the inversion function of Permute, and shares the same recursion tree, its depth is also bounded by $O(\log N)$ with high probability. □

8. CONCLUSION AND FUTURE WORK

8.1 Conclusion

We propose a novel construction for a small-domain pseudo-random permutation. As asymptotics is the wrong metric to optimize for small-domain problems, we instead aim for optimal practical performance. Our construction achieves strong security, i.e., can withstand up to N queries from an adversary, and is by far the most efficient construction for 32-bit integers or smaller domains, i.e., ($N < 2^{32}$). In particular, our construction is 1,000 to 8,000 times faster than the best known construction achieving a comparable level of security.

8.2 Discussions on Timing Channel and Future Work

Just like many other cryptographic algorithms, timing attacks can be a serious concern depending on how the algorithm is used. An algorithm can withstand timing attacks

if every pair of operation that the adversary wishes to distinguish between take the same amount of time to execute.

In our algorithm, the bitstring scanning operations dominate the execution time of the Permute and Unpermute algorithms. Therefore, in order to defend against timing attacks we need to scan the same number of bits regardless of the inputs. This can easily be done by performing additional *dummy scans* so the expected number of bits scanned equals the maximum number. If we don't perform counter alignment and we use bidirectional scanning, the expected amount of scanning for each invocation of C_1 doubles from $s/2$ to s, introducing a 2X slowdown in the Permute and Unpermute algorithms. With counter alignment present, the maximum number of bits scanned becomes $2s$, but it can be reduced to s by introducing an additional cached counter for the possible input ranges of C_1 at depths $d = 0, 1, \ldots, \log_2(N/s)$. When $s \in O(\sqrt{N})$, this increases the key size by a small factor of $O(1/\log N)$.

Timing difference can also be observed due to the variation in the depth of the Permute and Unpermute algorithms. Similarly, a padding idea can be applied to hide such variation. In future work, we plan to investigate exactly how much performance penalty will be incurred to perfectly defend against timing channel attacks.

Note that in many cases, it may not be necessary to modify the algorithm to defend against timing attacks. The running times of Permute and Unpermute are many orders of magnitude lower than typical network latency, so often times the fluctuations in execution time will be dominated by fluctuations in network latency.

9. REFERENCES

[1] Bit Twiddling Hacks. `http://graphics.stanford.edu/~seander/bithacks.html`.

[2] The Case for Elliptic Curve Cryptography. `http://www.nsa.gov/business/programs/elliptic_curve.shtml`.

[3] Tiny encryption algorithm. `http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm`.

[4] Xtea. `http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm`.

[5] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, pages 295–312, 2009.

[6] M. Bellare, P. Rogaway, and T. Spies. The ffx mode of operation for format-preserving encryption. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec.pdf`.

[7] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *CT-RSA*, pages 114–130, 2002.

[8] M. Brightwell and H. E. Smith. Using datatype-preserving encryption to enhance data warehouse security. In *National Information Systems Security Conference (NISSC)*, 1997.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[10] L. Granboulan and T. Pornin. Perfect block ciphers with small blocks. In *FSE*, pages 452–465, 2007.

[11] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, Apr. 1988.

[12] U. Maurer and K. Pietrzak. The security of many-round luby-rackoff pseudo-random permutations. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*, EUROCRYPT'03, pages 544–561, Berlin, Heidelberg, 2003. Springer-Verlag.

[13] B. Morris, P. Rogaway, and T. Stegers. How to encipher messages on a small domain. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '09, pages 286–302. Springer-Verlag, 2009.

[14] J. Patarin. Luby-rackoff: 7 rounds are enough for $2^{n(1\text{-epsilon})}$security. In *CRYPTO*, pages 513–529, 2003.

[15] J. Patarin. Security of random feistel schemes with 5 or more rounds. In *CRYPTO*, pages 106–122, 2004.

[16] V. Pryamikov. Enciphering with arbitrary small finite domains. In *INDOCRYPT*, pages 251–265, 2006.

[17] R. Schroeppel. Hasty pudding cipher specification. `http://richard.schroeppel.name:8015/hpc/hpc-spec`.

[18] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. In *NDSS*, 2012.