Abelian square-free dithering for iterated hash functions

Ronald L. Rivest

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
rivest@mit.edu

Draft of August 14, 2005

Abstract. We present a novel way of "dithering" the operation of an iterated hash function, based on the notion of *square-free* sequences (sequences containing no consecutive repeated subsequences) and on their generalization to abelian square-free sequences, as a means of defeating some "message expansion" attacks recently proposed by Dean and of Kelsey and Schneier [Dea99,KS05].

1 Introduction

This work is motivated by the thesis of Dean [Dea99] and the more recent paper of Kelsey and Schneier [KS05] on certain message-expansion methods that yield surprisingly efficient second-preimage attacks on iterated hash functions.

We begin here with a general overview of iterated hashing, to set the stage and define notation. A hash function is given as input a message M of some arbitrary nonnegative length L over some fixed alphabet Σ. That is,

$$M = M_0 M_1 \ldots M_{L-1}$$

where each M_i is an element of Σ. The hash function H is to produce as output a value $H(M)$ drawn from some finite set R; typically R consists of the set $\{0,1\}^n$ of all strings of some fixed length n. The range R does not depend on the length L of the input message.

In some formulations, the hash function takes as an auxiliary input a *key* K, in which case we have a *family* $\mathcal{H} = \{H_K\}$ of hash functions. We denote the set of possible keys by $\mathcal{K}$. The key may or may not be kept secret from an adversary.

The hash function should be efficiently computable: given K and M it should be possible to compute $H_K(M)$ in polynomial time. Since hash functions are often used on very long messages, it is desirable that the hash function be computable in *linear* time (i.e. time $O(L)$).

Since it may be desired that a hash function be computable by devices with limited storage, it is furthermore generally desirable that a hash function be computable not only in linear time, but also in an *on-line* manner: the computation of $H(M)$ should process each input symbol M_i in turn. This captures the idea of an *iterated hash algorithm*, an idea that first appeared in the seminal works of Rabin [Rab78,Rab79]. (Rabin used successive message blocks as keys to an iterated encryption operation using a block cipher.)

The most common approach for designing an iterated hash algorithm, used in many practical hash function designs, such as MD5 and SHA1, is the "Merkle-Damgard" framework [Mer90,Dam90], which constructs a collision-resistant hash function that processes arbitrary-sized inputs by iterating a collision-resistant "compression function" f that takes fixed-sized inputs. The collision-resistance of the hash function so constructed follows from the collision-resistance of the compression function.

Definition 1 (Iterated hash algorithm). *An iterated hash algorithm takes as input a key $K \in \mathcal{K}$ and a message $M = M_0 M_1 \ldots M_{L-1}$ of length $L \geq 0$ over some finite alphabet Σ, and produces an output $H(M)$ from some finite set R, in the following manner:*

1. **Set initial state.** *An initial state $h_{-1} = h_{-1}(K)$ is determined; this value h_{-1} is called the* initialization vector*, or IV.*
2. **Recode input** *The input M is* recoded *to*

$$M' = M'_0 M'_1 \ldots M'_{L'-1}$$

where each symbol M'_i is from some alphabet Σ'. In practice, such recoding generally involves just appending a one and then some zeros (to get the length to be a suitable length) and then appending the length of M in bits (this is Merkle-Damgård *strengthening).*

3. **Process each symbol of recoded input.** *For each i, $0 \leq i < L'$, the next state is computed:*

$$h_i = f(h_{i-1}, M'_i) \tag{1}$$

where f is some function (called a compression function*) The state values h_{-1}, h_0, h_1, $\ldots h_{L'-1}$ are also called* chaining variables.

4. **Produce output** *For some* output function *g, the output value $H_K(M)$ is computed as $g(K, h_{L'-1})$. (Sometimes the output function g is just the identity function.)*

Section 2 discusses the general notion of dithering an iterated hash function. Section 3 briefly reviews another framework, based on recoding. Section 4 then reviews the notions of square-free words and abelian square-free sequences, and shows that they can be efficiently generated. Section 5 discusses how an abelian square-free sequence can be used to dither an iterated hash function. Section 6 gives a concrete proposal for dithering an iterated hash function. Section 7 presents efficient ways of generating square-free and abelian square-free sequences. Section 8 gives a couple of miscellaneous facts about square-free sequences. Section 9 concludes with some discussion and open problems.

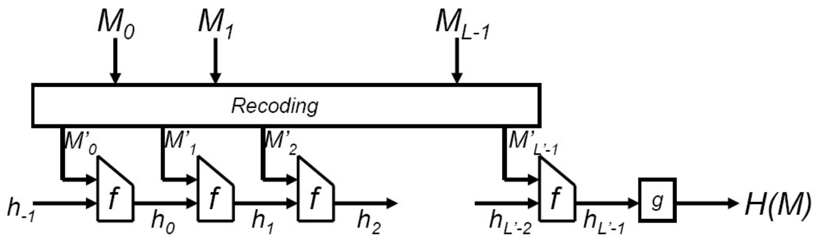

Fig. 1. The structure of an iterated hash function. The message $M = M_0 M_1 \ldots M_{L-1}$ is first recoded, yielding $M' = M'_0 M'_1 \ldots M'_{L'-1}$. An initialization vector $h_{-1} = IV$ is chosen. Each recoded message element M'_i in turn is combined with the previous chaining variable h_{i-1} using the *compression function* f to yield the next chaining variable h_i. The final output, the hash $H(M)$ of the input M, is obtained by applying an *output function* g to the final chaining variable.

2 Hash function dithering

In this section we discuss the notion of hash function dithering.

Dithering provides opportunities for improvements that defeat some known and some recently suggested attacks on hash functions. In particular, it rather simply defeats attacks [Dea99,KS05] based on *message block repetition* and generalizations thereof.

The word "dithering" derives from image-processing, where a variety of gray or colored values can be represented by mixing together pixels of a small number of basic shades or colors; this is done in a random or pseudo-random manner to prevent simple visual patterns from being visible. We adapt the term dithering here to refer to the process of adding an additional "dithering" input to a sequence of processing steps, to prevent an adversary from causing and exploiting simple repetitive patterns in the input.

The Ph.D. thesis of Dean [Dea99], and the more recent paper of Kelsey and Schneier [KS05], show how an adversary might be able to employ message block repetition to advantage. Their attacks take advantage of fixpoints of the compression function to derive second pre-images in a surprisingly efficient manner. A fixpoint is defined as a pair (h_{i-1}, M'_i) such that (referring to equation (1))

$$h_i = h_{i-1} = f(h_{i-1}, M'_i) \ .$$

Finding a fixpoint allows an adversary to repeat a given message block an arbitrary number of times, leaving the chaining variable unchanged; this allows the adversary to defeat Merkle-Damgard strengthening, since the adversary can use the fixpoint to expand the message appropriately, and thus more easily find a second message of the same length as the first message having the same hash value.

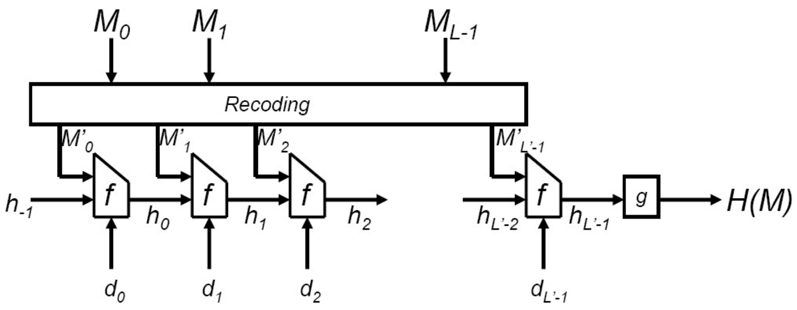

Fig. 2. The structure of an iterated hash function with dithering. Compare with Figure 1. The compression function f now takes an additional "dither" input d_i. For example, we may have a dither sequence $d = d_0 d_1 \cdots = \mathbf{z} = \texttt{abcacdcbcdcadcdb}\ldots$, Keränen's abelian square-free sequence.

These attacks allow an adversary to find a second pre-image in time

$$t2^{n/2+1} + 2^{n-t+1}$$

given an n-bit hash value for a first message of length 2^t blocks.

One can view these attacks as providing support for the hypothesis that the adversary (choosing the message) has *too much control* over the message.

How might one employ dithering to defeat such message expansion attacks? Several possibilities come to mind:

- [**Dithering with a counter**] One could dither by using the index i as the dither value:

  $$d_i = i\ . \tag{2}$$

  This was suggested by Kelsey and Schneier. This should work, but requires that the compression function f accept an arbitrarily large input i (since we don't wish to bound the size of the message input M, i can grow arbitrarily large). One would like the dithering elements to be elements from some small finite alphabet.

  It is not terribly difficult to design a compression function to take as an additional input a large (e.g. 160-bit) dither input, in which case the counter-based approach of equation (2) can be used directly, even for very long messages.

  Nonetheless, we feel that it is valuable to explore what can be accomplished using only a small finite alphabet for the dither input. Making the dither input more compact may also yield improvements in efficiency, compared to the counter-based approach.

  Note that Biham [Bih] proposes using the number of bits processed so far as an auxiliary input, rather than the index i of the block. This removes the need for Merkle-Damgård strengthening, but requires even more input bit positions for this counter into the compression function.

- [**Dithering with a pseudorandom sequence**] One could dither by utilizing some pseudo-random sequence $r_0, r_1, \ldots$:
$$d_i = r_i \ .$$
This only provides weak protection against message block repetition. As the message grows longer and longer, we expect to see longer blocks of repetitions among the d_i's. It is possible to do better.
- [**Dithering with alternating 0's and 1's**] One could dither by utilizing a sequence of alternating 0's and 1's:
$$d_i = \begin{cases} 0 & \text{if } i \text{ is even} \\ 1 & \text{otherwise.} \end{cases}$$
This protects against repetition of single message blocks, but doesn't protect against repetition of pairs of blocks, since the dither input now has period two. One would wish to have protection against repetitions of any segment of message blocks, since an adversary might find fixpoints based on repeating an arbitrary number of message input blocks.

If the goal is to prevent the adversary from *repeating* certain inputs (or certain input sequences), then the best approach may be, as sugested here, to use *square-free sequences*. These are aperiodic sequences over a finite alphabet with the property that no subword is repeated.

We note that the "dither input" might also be used for other purposes, such as to distinguish the computation of the last block. See Section 6 for an example. This usage makes the sequence of inputs prefix-free, which has beneficial effects (see [CDMP05]).

3 Recoding

We briefly discuss recoding, which provides an alternative avenue for incorporating variability.

Recoding should be efficient in an "on-line" sense, although Definition 1 doesn't specify any such restriction, to allow approaches that may be somewhat less efficient, but attractive from a security viewpoint.

To ensure that recoding isn't carrying the burden of achieving one-wayness or collision-resistance, we require that recoding be one-to-one and efficiently invertible without knowledge of any secret key.

Recoding can serve a number of purposes.

One such purpose may be to recode the message so that its length in bits becomes a multiple of some desired block size (say by appending to the message first a 1 and then sufficiently many 0's to make its length such a multiple).

A second purpose is often to defeat various message extension attacks by appending a length count (as in *Merkle-Damgard strengthening*) [Mer90,Dam90].

A third such purpose may be to provide a *minimum distance* property, so that a small change in the message M is guaranteed to produce a large change in the recoded message M'. This may be accomplished using error-correcting codes. We do not explore this direction further here, but refer the reader to e.g. Knudsen and Preneel's recent paper [KP02] for some treatment of error-correction in hash functions. We note that the message expansion in SHA-1 [NIS93] is an example of such minimum-distance recoding.

A fourth such purpose, presented in section 2, is to introduce some time-dependent variability during recoding, so that the way in which the input message M is converted into the recoded message M' changes and evolves as the input message is processed. We call this high-level recoding concept *dithered message recoding*. However, for the purposes of this paper we shall assume a more straightforward approach of having separate dithering inputs to the compression function; the functionality obtained is equivalent.

4 Square-free and abelian square-free words

We next give an overview of square-free sequences, and give an example of an infinite square-free word: the Thue sequence. We then examine a stronger form: abelian square-free sequences, and given an example of an infinite abelian square-free word: the Keränen sequence.

Although these sequences can be easily generated efficiently, we defer the discussion of such efficient generation until Section 7.

We prefer the use of abelian square-free sequences for our application, as they are just as easy to generate, and are "repetition-free" in a stronger sense than ordinary square-free sequences.

Basic definitions

A *word* is a (finite or infinite) sequence of letters over some finite alphabet. A word w is a *square* if it is of the form yy for some finite nonempty word y. If a word w can be written in the form xyz for words x, y, and z (where y is nonempty, but x and/or z may be empty), then we say that y is a *subword* of w. Some authors prefer to say that y is a *factor* of w.

Square-free words

A word w is said to be *square-free* if it contains no repeated subword: not only does w contain no repeated letters, but it contains no repeated subword of any (finite nonzero) length. Square-free words are thus nonrepeating and nonperiodic in a strong way.

Definition 2 (Square-free words). *A word w is said to be* square-free *if it contains no squares; that is, if w contains no subword of the form yy where y is finite and nonempty.*

Thus, `tomato` is square-free but `banana` (double `an`) is not.

It is easy to show that no square-free words of length greater than three exist over a *binary* alphabet.

An easy-to-generate infinite square-free sequence **u** over three letters is due to Thue (see Section 7.2):

$$\mathbf{u} = \texttt{210201210120210201202101210201210120}\ldots \; . \tag{3}$$

Abelian square-free words

Although a square-free sequence meets our original design goal of no repetitions of single symbols or of longer subwords, there are sequences meeting a stronger requirement—that of being *abelian* square-free—that are no harder to generate. The notion of being abelian square-free is stronger in that every abelian square-free sequence is also square-free in the ordinary sense. Because abelian square-free sequences are even more "repetition-free" than ordinary square-free sequences, and because they are no harder to generate, we recommend the use of abelian square-free sequences for dithering iterated hash functions.

Definition 3 (Abelian square-free words). *A word w is said to be* abelian square-free *if it can not be written in the form $w = xyy'z$ for words x,y,y', z, where y is not the empty word and where y' is a permutation of y.*

For example, `abcdcbda` is square-free but not abelian square-free, since the subword `bcd` is followed by its permutation `cbd`. Clearly every sequence that is abelian square-free is also square-free, since the permutation relating y' to y may be the identity permutation.

T. C. Brown [Bro71] provides a survey of known (in 1971) results on abelian square-free words, and gives as an open research problem to find an infinite sequence on a finite alphabet that is abelian square-free. Pleasants [Ple70] provided the first proof that infinite abelian square-free words exist over a finite (5-letter) alphabet.

Keränen [Ker92,Ker03] provides an elegant solution to the problem of finding an infinite abelian square-free sequence over a four-letter alphabet; this easy-to-generate sequence begins (see Section 7.8):

$$\mathbf{z} = z_0, z_1, \ldots = \texttt{abcacdcbcdcadcdbdabacabadbabcbdbcba}\ldots \; . \tag{4}$$

(See also Figure 7.)

5 Dithering using abelian square-free sequences

Keränen's sequence **z** can be used directly as a dither sequence. The compression function f needs to be designed to take as an additional input the two-bit "dither" input d_i from the Keränen sequence. The compression function f may treat the dither input essentially as if they were additional message input bits.

Another approach can be used if the one-way compression function is built around a *tweakable block cipher* [LRW02,BCS05,HL05]: one can use the dithering input to the compression function as the "tweak" input to the block cipher.

In the next section, we give a concrete proposal for dithering based on Keränen's abelian square-free sequence.

6 A concrete proposal for dithering iterated hash functions

We now give a concrete proposal for dithering iterated hash functions, to exemplify further our ideas and give some ideas for efficient dithering.

The scheme utilizes the abelian square-free sequence $\mathbf{z}$ over four letters due to Keränen, which can be efficiently generated as shown in Section 7. Each symbol of this sequence will be encoded as a two-bit value b, with the natural encoding: $\mathtt{a} \rightarrow 00$, $\mathtt{b} \rightarrow 01$, $\mathtt{c} \rightarrow 10$, $\mathtt{d} \rightarrow 11$.

Each dither input d consists of three values (a, b, c). The dither input contains a mod-2^{13} block counter c, and the element b of the Keränen sequence is only advanced once every 2^{13} blocks. The last block is treated specially: the last-block flag bit a is set to 1, the other dither input bits (b and c combined) indicate how many valid message bits are present in this block, and unused message input bits are set to zero.

This scheme thus has the following advantages:

1. The abelian square-free sequence is only generated at "rate 1/8192"; usually the dither input is just obtained from the previous dither input by incrementing the counter c. Only when c "overflows" does the abelian-square free sequence need to be advanced. Even though Keränen's sequence can be generated very efficiently, the approach here makes dither input generation exceptionally fast (usually just a counter increment). Including the mod-2^{13} counter as part of the dither input maintains the property that the dither input sequence is square-free.
2. The use of a special last-block flag bit a makes the input encoding effectively prefix-free. Knowing the hash output for a message M thus does not allow anyone to compute the hash output for an extension of M. Coron et al. [CDMP05] prove that such prefix-free encodings are effective at improving hash function security.
3. The use of the special last-block flag allows one to eliminate the need for Merkle-Damård strengthening. That is, there is now no need to "pad" the message by adding a one, some zeros, and the length of the message. The use of the dither input for the last block to indicate the number of valid message bits in that last block allows one to handle messages of arbitrary bit-length, even though the compression function may only take a fixed-length message input.

To be precise, each dither input d_i, $i \geq 0$, is two bytes (16 bits) in length, and consists of (see Figure 3):

1. A one-bit *last-block flag bit a*. This bit is normally 0, but is 1 for the last block processed.
2. A two-bit value b that is normally the $\lfloor i/8192 \rfloor$-th element z_i of the Keränen sequence, for the i-th message block, $i \geq 0$. For the last block, b and c form a 15-bit value indicating the number of valid message bits in this block.

3. A 13-bit counter c which is normally $i \pmod{8192}$ for the i-th message block, $i \geq 0$. For the last block, b and c form a 15-bit value indicating the number of valid message bits in this block.

For the last block, the value indicated by the 15-bit combined value bc must be positive, unless the last block is also the first block and the total message has length 0 bits, in which case $bc = 0$. In any case, unused compression function message input bits must be zero.

Some free C code for generating this proposed dither sequence is available on my web site [Riv05].

This proposal would need revision if the compression function took as input a message block of more than $2^{15} - 1 = 32767$ bits; the value c could be expanded by a byte or two in a natural manner in that case. However, most compression functions today only take 512-bit input message blocks.

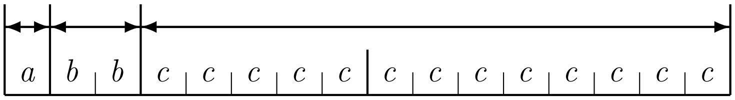

Fig. 3. A two-byte dither input with a as the high-order bit of the first byte. For block i, $i \geq 0$, other than the last block, a is 0, b encodes the $\lfloor i/2^{13} \rfloor$-th element of Keränen's sequence, and c is $i \pmod{2^{13}}$. For the last block, $a = 1$, and bits bc form a 15-bit unsigned integer giving the number of message bits processed in this block.

Existing hash function designs such as MD5 or SHA-nnn can easily be modified to accommodate a dither input. Instead of breaking the message into 512-bit (64 byte) blocks, it is instead broken into 62-byte blocks, and the two-byte dither input is appended to the end of each 62-byte block. Including the dither input as two bytes of the message input block entails a small performance loss of approximately 3.2%.

7 Efficient Generation of Square-Free and Abelian Square-Free Sequences

We now show that square-free and abelian square-free sequences can be generated efficiently. (This is perhaps not so terribly important, at least in the context of the concrete proposal of Section 6, where the next symbol needs to be generated only once per every 8192 message blocks.)

The material in this section is generally a restatement of material already in the literature; for more information see Allouche and Shallit [AS03], as well as papers cited in their on-line bibliography [AS]. The results on abelian square-free sequences are due to Keränen [Ker92,Ker03].

We start with a presentation of the Prouhet-Thue-Morse (PTM) sequence **t**, which is cube-free but not square-free (Section 7.1). We move on to the Thue sequence **u**, which is derived from **t** and which is square-free (Section 7.2). In Section 7.3 we explain how infinite sequences can be defined as a limiting fixpoint of a homomorphism. Section 7.4 then shows how the i-th element of such an infinite sequence can be computed by running a finite automaton over the binary (or k-ary) representation of i. Section 7.5 explains how each successive symbol in such a sequence can be produced in amortized constant time–this is really little more than the corresponding observation that incrementing a k-ary counter can be done in constant amortized time. Section 7.6 describes an alternative way of generating square-free sequences. In Section 7.7 we present the "Towers of Hanoi" infinite square-free sequence due to Shallit, which is interesting because each symbol can easily be generated in *worst-case* constant time. Finally, in Section 7.8, we carefully define Keränen's infinite abelian square-free sequence, and show that it can be generated efficiently.

7.1 Prouhet-Thue-Morse sequence

In this subsection we define the famous Prouhet-Thue-Morse (PTM) sequence **t**; it is also called the *Thue sequence* or the *Thue-Morse sequence*.

Although it is not square-free, it is cube-free, and can be used to construct square-free sequences in several ways. This sequence **t** is defined over the binary alphabet $\{0, 1\}$:

$$\mathbf{t} = t_0, t_1, \ldots, = \texttt{011010011001011010010110011010011001011001101001...}$$

The i-th bit t_i of **t** can be defined in many ways; perhaps the simplest is in terms of the parity function applied to the binary representation of i, for $i = 0$,1, ...:

$$t_i = \begin{cases} 0 & \text{if the binary representation of } i \text{ has an even number of ones} \\ 1 & \text{otherwise.} \end{cases} \tag{5}$$

7.2 Thue sequence

In 1906, Axel Thue exhibited an infinite *square-free* word **u** over a ternary alphabet. It begins:

$$\mathbf{u} = u_0, u_1, \ldots = \texttt{210201210120210201202101210201210120}\ldots \tag{6}$$

The i-th element of this sequence, for $i \geq 0$, can be defined as the number of ones occurring between the i-th and $i + 1$-st zeros in the PTM sequence (where t_0 is the 0-th zero). Since **t** is cube-free, the number of times a one can be repeated is at most two, so that **u** is defined over the ternary alphabet $\{0, 1, 2\}$.

7.3 Morphisms

In this subsection we define the general notion of a morphism, which can be used to define various square-free words. Let Σ be some fixed finite alphabet, let Σ^* denote (as usual) the set of all finite-length strings over Σ, and let ϵ denote the empty string.

A *morphism* (short for *homomorphism*) is a mapping τ from Σ to Σ^*, extended to a mapping from Σ^* to Σ^* by defining $\tau(\epsilon) = \epsilon$, and defining

$$\tau(xy) = \tau(x)\tau(y)$$

for nonempty words x, y. A morphism is said to be *k-uniform* if $|\tau(a)| = k$ for all $a \in \Sigma$. A morphism is said to be *non-erasing* if $\tau(a)$ is nonempty for all $a \in \Sigma$.

Consider the 2-uniform morphism

$$\tau(\mathtt{0}) = \mathtt{01}, \quad \tau(\mathtt{1}) = \mathtt{10}\ . \tag{7}$$

Then we see that iterating this morphism starting with `0`

$$\mathtt{0} \xrightarrow{\tau} \mathtt{01} \xrightarrow{\tau} \mathtt{0110} \xrightarrow{\tau} \mathtt{01101001} \xrightarrow{\tau} \mathtt{0110100110010110} \xrightarrow{\tau} \ldots \xrightarrow{\tau} \tau^\infty(\mathtt{0}) = \mathbf{t}$$

defines the infinite PTM sequence $\mathbf{t}$ in the limit as the infinite fix-point of the mapping τ beginning with `0`. In general, if τ is non-erasing, and for some $a \in \Sigma$, $\tau(a)$ begins with a, then there exists an infinite word $\tau^\infty(a)$ that is the unique fixpoint of τ that begins with a.

As another example, the Thue's square-free sequence $\mathbf{u}$ can be defined by starting with the 2-uniform morphism

$$\tau(\mathtt{A}) = \mathtt{AB}, \quad \tau(\mathtt{B}) = \mathtt{CA}, \quad \tau(\mathtt{C}) = \mathtt{CD}, \quad \tau(\mathtt{D}) = \mathtt{AC}$$

over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$. This has a unique fixpoint starting with `A`:

$$\tau^\infty(\mathtt{A}) = \mathtt{ABCACDABCDACABCA}\ldots\ ,$$

and then applying to this result the morphism

$$\mu(\mathtt{A}) = \mathtt{2}, \quad \mu(\mathtt{B}) = \mathtt{1}, \quad \mu(\mathtt{C}) = \mathtt{0}, \quad \mu(\mathtt{D}) = \mathtt{1}\ .$$

7.4 Finite automata

There is a close relationship between the two representations we have given above for the PTM sequence $\mathbf{t}$—as the limit of the morphism (7) and the definition (5) in terms of parity of binary representations. Suppose a string $\mathbf{v}$ is defined as the limit point $\tau^\infty(\mathtt{a})$ of the k-uniform morphism τ over the alphabet Σ. Then the i-th element v_i of $\mathbf{v}$ can be defined as the result of running a simple finite automaton on the k-ary representation of the integer i (high-order digit first).

The state set of the automaton is the alphabet Σ, and the start state is the symbol `a`. The input alphabet is the set $\{0, 1, \ldots, k-1\}$. The transition function from state a on input j, for $0 \leq j < k$, is just the j-th symbol in the string $\tau(a)$ (using 0-origin indexing of strings). The output v_i of the finite automaton is some function μ of the final state reached after the least-significant k-ary digit of the integer i is processed.

Figure 4 illustrates this approach for generating the Thue's infinite square-free sequence **u**; for this sequence there is a separate output function. One can generate the i-th element of the sequence (6) by running the binary representation for i (high-order bit first) through the automaton of Figure 4, and taking the output associated with the final state.

state ↓ input →	0	1
A	A	B
B	C	A
C	C	D
D	A	C

state	output
A	2
B	1
C	0
D	1

Fig. 4. The next-state and output functions for a finite automaton useful in producing the infinite square-free Thue sequence 2102012... of equation (6). The start state is A. The automaton accepts the bits of the binary representation of the integer i, $i \geq 0$, high-order bit first. The output associated with the final state is the i-th element of the Thue sequence **u**.

7.5 Constant-time (amortized) generation

Using the above approach, and with a little extra care, each successive element of a sequence defined as the limit point of a nonerasing k-uniform morphism can be produced in *constant time in an amortized sense* (see [CLRS01, Chapter 17] and Appendix A of this paper). We now show how each output symbol can be produced in constant amortized time.

Let $(\ldots, c_3, c_2, c_1, c_0)$ denote the k-ary representation of the integer i:

$$i = \sum_{j \geq 0} c_j k^j \ .$$

Let $(\ldots, s_3, s_2, s_1, s_0)$ denote the states the finite automaton is in; s_j is the sequence after processing c_j. See Figure 5 for efficient pseudocode.

7.6 Jensen et al.'s method

Jensen et al. [JKS04] have presented some alternative approaches for generating square-free sequences from the Prouhet-Thue-Morse sequence **t**. For example, they show that the sequence of pairs $(t_i, t_{i+\delta})$, for $i \geq 0$, is square-free over a four-letter alphabet, whenever δ is odd. Note that the case $\delta = 1$ is particularly easy for implementation.

```
$Gen()$
▷ Increment $k$-ary counter $c$ by one.
$j = 0$
while $c_j = k - 1$
      do $c_j = 0$
         $j = j + 1$
$c_j = c_j + 1$
▷ Now update state variables for changed digits.
while $j \geq 0$
      do $s_j = \tau(s_{j+1})[c_j]$
         $j = j - 1$
▷ return function $\mu$ of final state
return $\mu(s_0)$
```

Fig. 5. Pseudo-code for producing a sequence defined as the successive outputs when running a finite-state automaton on the successive k-ary representations of the integers. This pseudocode runs amortized constant time per symbol generated.

7.7 Towers of Hanoi sequence

Another cute approach (suggested by Jeff Shallit) for generating an infinite square-free sequence, on a a six-letter alphabet, is to solve optimally the "Towers of Hanoi" with d disks on 3 pegs, recording each peg to peg move. Let us call the pegs X, Y, and Z. There are six possible moves. The optimal sequence moves the smallest disk on each odd-numbered move cyclically from peg X to Y to Z to X etc.; the even-numbered moves are then forced. See Figure 6. The resulting sequence

$$\texttt{ABDAEFABDC}\ldots \tag{8}$$

is square-free; this result has been proven by Allouche et al. [AARS94]; see also Allouche and Shallit [AS03]. It is easy to produce an implementation for large (essentially infinite) d.

The Towers of Hanoi sequence is interesting from an implementation viewpoint, since it is the only square-free sequence I know of that allows each output symbol to be easily produced in *worst-case constant time*. The other sequences seem much more difficult to produce in amortized constant time per output symbol. Perhaps there are optimization techniques I have overlooked that allow each output symbol to be easily produced in worst-case constant-time for these other sequences as well. Of course, the fact that a constant amortized cost procedure exists implies that a worst-case cost procedure exists as well, but the buffering and scheduling required are not simple.

Move	Peg X	Peg Y	Peg Z
Start state	54321	-	-
A (X→Y)	5432	1	-
B (X→Z)	543	1	2
D (Y→Z)	543	-	21
A (X→Y)	54	3	21
E (Z→X)	541	3	2
F (Z→Y)	541	32	-
A (X→Y)	54	321	-
B (X→Z)	5	321	4
D (Y→Z)	5	32	41
C (Y→X)	52	3	41
...	...	...	...

Fig. 6. The square-free sequence ABDAEFABDC... describing optimal "Towers of Hanoi" play. The pegs are labelled X, Y, and Z. The disks are labelled from 1 (smallest) to 5 (largest); a disk may never be placed on top of a smaller disk. Here A represents a move from peg X to peg Y, B: $X \to Z$, C: $Y \to X$, D: $Y \to Z$, E: $Z \to X$, and F: $Z \to Y$.

7.8 Keränen's infinite abelian square-free sequence

Keränen's sequence (see Figure 7) is very easy to generate efficiently (constant time on the average per output symbol), as explained in Figure 7, and also in the code in Appendix A.

Let $s_{\texttt{a}}$ denote the following magic sequence of length 85 over the alphabet $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$:

$$s_{\texttt{a}} = \texttt{abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca}$$

Let $s_{\texttt{b}}$, $s_{\texttt{c}}$, and $s_{\texttt{d}}$ denote the result of applying a single, double, or triple cyclic shift of the alphabet to $s_{\texttt{a}}$:

$$s_{\texttt{b}} = \texttt{bcdbdadcdadbadacabcbdbcbacbcdcacdcbdcdadbdcbcabcbdbadcdadbdacdcbdcdadbdadcadabacadcdb}$$

$$s_{\texttt{c}} = \texttt{cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac}$$

$$s_{\texttt{d}} = \texttt{dabdbcbabcbdcbcacdadbdadcadabacabadbabcbdbadacdadbdcbabcbdbcabadbabcbdbcbacbcdcacbabd}$$

Then Keränen's infinite sequence is obtained by beginning with the single letter a, and then repeatedly applying the morphism τ defined by:

$$\tau(\texttt{a}) = s_{\texttt{a}}, \quad \tau(\texttt{b}) = s_{\texttt{b}}, \quad \tau(\texttt{c}) = s_{\texttt{c}}, \quad \tau(\texttt{d}) = s_{\texttt{d}}$$

simultaneously to all letters. Because $s_{\texttt{a}}$ begins with an a, the sequence of words a, $\tau(\texttt{a})$, $\tau^2(\texttt{a})$, ..., converges in the limit to an infinite abelian square-free word **z**, whose initial 1700 symbols are given in Figure 7. It is rather amazing that no word shorter than 85 symbols on four letters can generate an infinite abelian sequence in this way, although a 15-letter word on five letters can do so [Ker03,Ple70].

In practice, an efficient generation procedure can be based around a procedure that counts modulo 85, and runs the resulting base-85 representation through an appropriate finite-state automaton. See Appendix A.

A C implementation on an IBM T42p laptop (with a 1.8GHz Pentium M 745 processor) generates about 90M output values per second. This can be compared with a SHA-1 processing rate of about 470K (64-byte) input blocks per second.

```
abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca
bcdbdadcdadbadacabcbdbcbacbcdcacdcbdcdadbdcbcabcbdbadcdadbdacdcbdcdadbdadcadabacadcdb
cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac
abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca
cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac
dabdbcbabcbdcbcacdadbdadcadabacabadbabcbdbadacdadbdcbabcbdbcabadbabcbdbcbacbcdcacbabd
cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac
bcdbdadcdadbadacabcbdbcbacbcdcacdcbdcdadbdcbcabcbdbadcdadbdacdcbdcdadbdadcadabacadcdb
cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac
dabdbcbabcbdcbcacdadbdadcadabacabadbabcbdbadacdadbdcbabcbdbcabadbabcbdbcbacbcdcacbabd
cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac
abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca
dabdbcbabcbdcbcacdadbdadcadabacabadbabcbdbadacdadbdcbabcbdbcabadbabcbdbcbacbcdcacbabd
cdacabadabacbabdbcdcacdcbdcdadbdadcadabacadcdbcdcacbadabacabdadcadabacabadbabcbdbadac
dabdbcbabcbdcbcacdadbdadcadabacabadbabcbdbadacdadbdcbabcbdbcabadbabcbdbcbacbcdcacbabd
bcdbdadcdadbadacabcbdbcbacbcdcacdcbdcdadbdcbcabcbdbadcdadbdacdcbdcdadbdadcadabacadcdb
dabdbcbabcbdcbcacdadbdadcadabacabadbabcbdbadacdadbdcbabcbdbcabadbabcbdbcbacbcdcacbabd
abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca
bcdbdadcdadbadacabcbdbcbacbcdcacdcbdcdadbdcbcabcbdbadcdadbdacdcbdcdadbdadcadabacadcdb
abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca
```

Fig. 7. The first 1700 symbols of Keränen's abelian square-free sequence [Ker92] **z**, arranged as 20 rows of 85 letters each. This sequence can be generated by starting with the letter a, then repeating as often as desired the operation of simultaneously replacing each letter x with the 85-letter string $s_{\mathtt{x}}$. In this figure, $s_{\mathtt{a}}$ appears as the first row, and $s_{\mathtt{x}}$ appears as the row beginning with letter x. We note that $s_{\mathtt{b}}$ is the same as $s_{\mathtt{a}}$ with each letter replaced by the one following it cyclically in the alphabet; $s_{\mathtt{c}}$ and $s_{\mathtt{d}}$ follow similarly.

8 Miscellaneous facts about square-free sequences

The preceding subsection shows how to generate Keränen's abelian square-free sequence, and to use it as a dithering sequence.

Another approach to dithering is to incorporate it into the recoding operation.

In a cross-product approach, we start with an infinite abelian square-free sequence $w = w_0 w_1 w_2 \ldots$ where each element w_i is a symbol from an alphabet Σ, and with a finite message sequence M_0, M_1, $\ldots$, where each element w_i is a symbol from an alphabet Σ'. The resulting encoding has the form: $M' = M'_0 M'_1 \ldots M'_{L-1}$ where $M'_i = (w_i, M_i)$. That is, the i-th element of the recoded input is the ordered pair consisting of w_i and M_i; this is a symbol from the cross-product alphabet $\Sigma \times \Sigma'$. It is trivial to see that this recoding procedure results in an word that not only represents M (i.e., M can be extracted from M'), but that is abelian square-free as well.

Another recoding approach uses a "perfect shuffle" to combine an arbitrary message sequence with a fixed (abelian) square-free sequence (such as Keränen's) yields a sequence that is both an encoding of the message and (abelian) square free. The theorem requires that the fixed (abelian) square-free sequence and the message sequence be on disjoint alphabets. It is easy to see that this condition is necessary.

As an example, the 16-letter word `HHHHAAAASSSSHHHH` can be perfectly shuffled with the first 16 letters of Keränen's sequence: `abcacdcbcdcadcdb` to yield: `HaHbHcHaAcAdAcAbScSdScSaHdHcHdHb`. This process can obviously be extended for as long as desired, since the Keränen sequence is infinite.

Theorem 1 ((Abelian) Square-free perfect shuffling theorem). *If $w = w_1w_2\ldots$ is an infinite square-free sequence (respectively abelian square-free) over an alphabet Σ and x_1, x_2, $\ldots$ is an infinite sequence of empty or finite square-free words (respectively empty or finite abelian square-free words)* on a disjoint alphabet *Σ', then the infinite sequence*

$$z = \textit{shuffle(w,s)} = w_1x_1w_2x_2w_3x_3\ldots$$

is square free (respectively abelian square-free) over the alphabet $\Sigma'' = \Sigma \cup \Sigma'$.

Note that the x_i's are words, not just symbols.

Proof. The word z is formed by alternating symbols from w with words from x. We prove the case for square-freeness; the proof for abelian square-freeness is almost identical. Let yy' be any subword of z, where $|y| = |y'|$. If yy' is a square, then the word obtained from yy' by erasing all symbols in Σ' must also be either empty or a square over the alphabet Σ. But the latter possibility contradicts the square-freeness of w. Therefore yy' must only contain symbols from Σ', which means it must be a subword of one of the x_i's. The theorem follows from the assumed square-freeness of the x_i's. □

9 Discussion, Open Problems, and Conclusions

We have presented a novel way of "dithering" an iterated hash function, to prevent attacks such as those of Dean [Dea99], and Kelsey and Schneier [KS05]. Our dithering approach is based on the use of square-free and abelian-square sequences.

We have also presented a particular concrete instantiation of this idea, using Keränen's infinite abelian-free sequence and a counter to produce a prefix-free square-free sequence of two-byte dither inputs.

As an open problem, we ask if there is an even stronger notion of "repetition-free" that would be usable for this application?

The flip side of this question is whether there is really anything being gained by using definitions of repetition-free that are stronger than square-free.

It would be interesting to find simple worst-case constant-time generation techniques for sequences defined by iterated morphisms; at present I know only of such a simple technique for the Towers Of Hanoi sequence.

Acknowledgments

I'd like to thank Jeff Shallit and Veikko Keränen for their generous help on square-free sequences and abelian square-free sequences.

References

AARS94. Jean-Paul Allouche, D. Astoorian, J. Randall, and J. Shallit. Morphisms, square-free strings and the Tower of Hanoi puzzle. *American Mathematics Monthly*, 101(7):651–658, August/September 1994.

AS. Jean-Paul Allouche and Jeffrey Shallit. (on-line bibliography on repetition-free sequences. `http://www.cs.uwaterloo.ca/~shallit/asbib/repetitions.bib`.

AS03. Jean-Paul Allouche and Jeffrey Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, 2003.

BCS05. John Black, Martin Cochran, and Thomas Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. In Ronald Cramer, editor, *Advances in Cryptology – Proceedings Eurocrypt '05*, volume LNCS 3494, pages 526 – 541. Springer, 2005.

Bih. Eli Biham. Recent advances in hash functions – the way to go. Presented at ECRYPT Conference on Hash Functions (Cracow, June 2005), see `http://www.ecrypt.eu.org/stvl/hfw/Biham.ps`.

Bro71. T. C. Brown. Is there a sequence on four symbols in which no two adjacent segments are permutations of one another? *American Math. Monthly*, 78:886–888, October 1971.

CDMP05. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology – Proceedings Crypto 2005*, volume XXXX, pages XXX–XXX. Springer, 2005. Lecture Notes in Computer Science.

CLRS01. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw Hill, 2001.

Dam90. Ivan B. Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology – Proceedings CRYPTO '89*, volume LNCS 435, pages 416–427. Springer, 1990.

Dea99. Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

HL05. Susan Hohenberger and Moses Liskov, 2005. Personal communication on novel construction of tweakable block ciphers; to appear.

JKS04. Erik Jensen, Veikko Keränen, and Klaus Sutner. Infinite sets of square-free ω-words derived from the Prouhet-Thue-Morse sequence. In *New*

Ideas In Symbolic Computation: Proceedings 6th International Mathematica Symposium (eProceedings on CD). Positive Corporation Ltd. (Hampshire UK), 2004. Also available at: `http://south.rotol.ramk.fi/keranen/IMS2004/IMS2004_ExtraMaterial.html` and `http://south.rotol.ramk.fi/keranen/IMS2004/PTMSequence_IMS2004PresentationByJKS.html`.

Ker92. V. Keränen. Abelian squares are avoidable on 4 letters. In W. Kuich, editor, *Proceedings 19th Conference on Automata, Languages, and Programming (ICALP)*, volume 623 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 1992.

Ker03. V. Keränen. On abelian square-free DT0L-languages over 4 letters. In T. Harju and J. Karhumäki, editors, *Proceedings 4th Conference on Combinatorics on Words*, volume 27 of *TUCS General Publication*, pages 95–109. Turku Centre for Computer Science, 2003. An earlier version is available at `http://south.rotol.ramk.fi/keranen/ias2002/NewAbelianSquare-FreeDT0L-LanguagesOver4Letters.pdf`.

KP02. Lars Knudsen and Bart Preneel. Construction of secure and fast hash functions using nonbinary error-correcting codes. *IEEE Transactions on Information Theory*, 48(9):2524–2539, September 2002.

KS05. John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Ronald Cramer, editor, *Advances in Cryptology – Proceedings Eurocrypt '05*, volume LNCS 3494, pages 474–490. Springer, 2005.

LRW02. Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology – Proceedings CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.

Mer90. R. C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *Advances in Cryptology – Proceedings CRYPTO '89*, volume LNCS 435, pages 428–446. Springer, 1990.

NIS93. NIST. Secure hash standard, May 11 1993.

Ple70. P. A. B. Pleasants. Non-repetitive sequences. *Proc. Cambridge Phil. Soc.*, 68:267–274, 1970.

Rab78. M. O. Rabin. Digitalized signatures. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 155–168. Academic Press, 1978.

Rab79. Michael Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1979.

Riv05. Ronald L. Rivest. C code for generating proposed dither sequence, 2005. `http://theory.lcs.mit.edu/~rivest/Rivest-AbelianSquareFreeDithering.c`.

Appendix A. Generating Keränen's abelian square-free sequence

```
# asf.py -- Python code to output Keranen's abelian square-free sequence.
# Ronald L. Rivest. June 12, 2005.

import string
# Magic sequence R is from:
#    V. Keranen, "Abelian squares are avoidable on 4 letters", Proc.
#    19th ICALP Conference, Springer LNCS vol. 623.  1992, pages 41--52.
R = "abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacbabdaba"
R = R + "cadcbcdcacdbcbacbcdcacdcbdcdadbdcbca"
k = len(R)
alphabet = "abcd"   # code assumes R and alphabet start with same letter.
m = len(alphabet)

# Works via a walk of a tree of branching factor k and height L.
# L is arbitrary; we just need  k**L > desired output length.
# Level L-1 is the root of the tree; level 0 has the leaves.
# Array C[0..L-1] gives state of walk, specifying path from leaf to root.
# 0<=C[i]<k for all i.  C is base-k counter, LS digit in C[0].
# Level i-th node is C[i]'th child of its parent (0-origin indexing).
# Each node has label between 0 and m-1, inclusive, stored in S, which is
# letter being expanded at that level; label of leaves seen are output.

def reset():
    """ Reset system to starting state. """
    global S, C, L
    L = 100; S = [0]*L; C = [0]*L   # arrays of L zeros

def out():
    """ Return output symbol for current state. """
    return alphabet[S[0]]

def inc():
    """ Move to next state. """
    # Increment counter in C by one, base k
    j = 0
    while C[j] == k-1: C[j] = 0; j = j + 1
    C[j] = C[j] + 1
    # Update state values for all levels that have changed count.
    while j >= 0: S[j] = ((ord(R[C[j]])-ord(R[0]))+S[j+1])%m; j = j - 1

def get(t):
    """ Return string with next t output symbols, advancing state. """
    answer = []
    for i in range(t): answer.append(out()); inc()
    return(string.join(answer,''))

reset(); for i in range(20): print get(k) # produce 1700 output symbols
```