Spiral Ratchet

A Recursive Hash System

Brooklyn Zelenka
Fission Codes
brooklyn@fission.codes

December 30, 2021

Abstract

This paper presents the spiral ratchet, a novel symmetric key derivation (KDF) chain that can be efficiently incremented by arbitrary intervals.

1 Introduction

Hash algorithms provide a way to generate deterministic but random-appearing data from an input that is impractical to reverse. Many applications iteratively hash a value ("ratcheting") for pre-computation resistance, for backwards secrecy, one-time schemes (e.g. S/KEY), deterministically generating unique names, key derivation functions (KDFs), and so on. We introduce the "spiral ratchet", a hierarchical iterative hashing system capable of efficiently making large leaps, while preserving backwards secrecy.

2 Motivation

The intersection of private data and location-independent access control in open distributed systems is only beginning to be explored. Trust minimized protocols are increasingly important, but admit to many unsolved problems, including how to secure changing data in an unknown and unstable topology.

Functional persistence is a common strategy in distributed data. The security of history is very important in many applications, including group messaging, collaborative documents, time-series data, and even streams of data with micropayments. It is often impractical to perform a random key rotation for each update. Simple linear ratchets are often used in these situations, such as iteratively applying SHA3-256. More sophisticated ratchet mechanisms can be employed when the participants are fixed to a small group, such as Signal's Double Ratchet.

An increasing number of applications have an unstable topology of peers. They require zero-interaction key agreement for arbitrary keys in a sequence, the ability to share a single update (and no others), a range of updates, or history from a point in time onwards. As such, there is a need to keep the historical information of the KDF internal state secret as well.

One approach for securing history is to use a simple ratchet function, iteratively hashing on each update to produce the next KDF state. This works well if the number of changes to synchronize is small, but $\mathcal{O}(n)$ ratchet generation steps is prohibitive as the difference grows. It allows a malicious participant to force others to perform the same amount of work.

The spiral ratchet adds several properties. It synchronizes in sublinear time, supports efficient arbitrary access, its internal state is backwards secret, and its internal state does not leak metadata such as the number of updates. It is configurable to permit jumps of any size and granularity, and is weakly break-in resistant.

3 Numeral Intuition

The problem of how to quickly calculate the nth step of a ratchet in less than n steps is structurally similar to how we efficiently represent large numbers. While not the only possible system, positional numeral systems are ubiquitous since they very elegantly and precisely express large numbers.

3.1 Unary Hashing

Unary is one of the simplest forms of representing numbers. This system is very concrete: the number of symbols is the number being represented. One example is tally marks, but the more computationally interesting are Peano numbers. These numerals are represented by a zero element 0, and successors S, such that the number 3 is represented $S(S(S(0)))$. This is very straightforward, but is neither space nor time efficient.

Peano numbers may expressed as functions via Church-encoded. Here, each successor is represented by function application. This model uses the constant identity function $\lambda f.\lambda x.x$ as "zero", and the application of a function to represent each successor, such that 3 is represented as $\lambda f.\lambda x.f(f(fx))$. Replacing the successor function with a hash function h, and an arbitrary initial value I as its zero, one may represent unary counting structurally identically to Peano numbers. In this context, 3 is represented as the hash chain $h(h(h(I)))$.

To avoid writing a large number of recursive applications by hand, let $h^n(I)$ be n recursive applications of h to I. By convention, this paper will use I, J, and K as independent initial values.

3.2 Positional Hashing

A major advantage of positionality is the ability to express large jumps with minimal effort. In positional systems, such as the familiar binary, decimal, and hexadecimal, each position corresponds to some factor for the numeral at that position. While most systems have factors that are some exponent of a fixed base, there are many systems where there are no simple relationship between factors.

$$246_{16} = 2{\times}16^2 + 4{\times}16^1 + 6{\times}16^0 = 582_{10}$$

Figure 1: Componentized Hexadecimal

By analogy, multiple values can placed in a tuple and iteratively hashed to represent digits in a compound structure.

$$\begin{aligned} h_{pos10}(0) &\Rightarrow I \\ h_{pos10}(11) &\Rightarrow \langle h(J), h(I) \rangle \\ h_{pos10}(582) &\Rightarrow \langle h^5(K), h^8(J), h^2(I) \rangle \end{aligned}$$

Figure 2: Compound Hash

Just as positional numerals combine multiple numbers to represent a single sum, a KDF can have multiple components that are combined to create a single symmetric key. Unlike a positional number system used in arithmetic, a KDF "forgets" its internal structure, combining elements and flattening its structure. There are many methods of combination, including further hashing; here we use binary XOR.

$$\begin{aligned} h_{pos10,\oplus}(0) &\Rightarrow I \\ h_{pos10,\oplus}(11) &\Rightarrow h(J) \oplus h(I) \\ h_{pos10,\oplus}(582) &\Rightarrow h^5(K) \oplus h^8(J) \oplus h^2(I) \end{aligned}$$

Figure 3: Compound Hash Key Derivation

While it is easier to reason about a consistent base, there are cases where the ability to skip by different intervals per level is useful. This paper will only consider the case where the base is consistent across all positional digits.

4 Bounding

Unlike number systems for arithmetic, the spiral ratchet has a fixed number of "digits" in its internal state. This limits the jumps to fixed intervals, with a maximum jump interval. It would be possible to deterministically generate increasingly large digits as needed, but (as we will see later) this would leak data about the range that the count is currently in, which is undesirable (e.g. for securing updatable documents.)

While ratcheting by arbitrary intervals is possible, the jump operation is not the same as adding arbitrary integers. Incrementing all but the lowest digit cascades down to the lower values, in effect "zeroing" them. Arbitrary jumps require a carry (Algorithm 4). This is structurally similar to skip lists (Figure 4), but with each step pointing to a monotonically increasing value, rather than previous values.

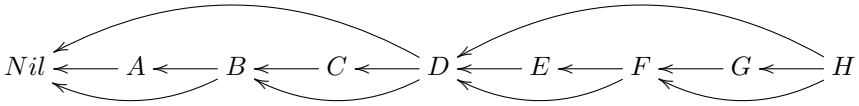

Figure 4: Skip List

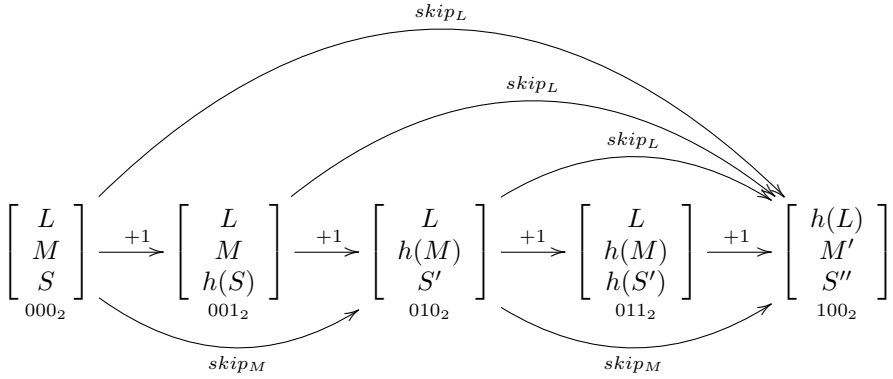

Figure 5: Simplified Compound Ratchet

Each digit is given a label and a maximum value for all but the largest, which acts as a linear spine. This treats the largest element as analogous to a unary digit, and the remaining elements are positional, bounded by their numerical base. The unary digit hides information about the range of numbers that the current internal state represents, which is called an "epoch".

This is roughly analogous to moving along a spiral: one component returns to its initial value, but the other is always moving forward. The motion is predictable and roughly circular, but never crosses itself. Extending the metaphor further: following the spiral in single steps is smooth, but leaps are possible by crossing to adjacent rings.

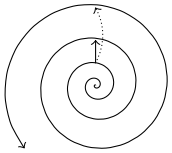

Figure 6: Spiral With Leaps

5 Spiral Ratchet

The spiral ratchet is built from a unary digit U, a fixed number n of positional digits, and their base b. The positional digits are bounded by the base, and so must track a natural number count. The unary digit explicitly does not track its count.

$$\langle\langle Count_0, Value_o\rangle, \langle Count_1, Value_1\rangle \dots \langle Count_{n-1}, Value_{n-1}\rangle, Unary\rangle_b$$

Figure 7: Spiral Ratchet State

Note that this is ordered as little-endian. The order of the state does not strictly matter, but it is convenient to associate the index of the state with the exponent for the base that the position represents.

5.1 Initialization

The base ratchet state is deterministically derived from a single initialization vector (IV). To prevent leaking the iteration count, all of the positional values are immediately incremented by a random value. This randomized origin is treated as the ratchet's initial value. Given the backwards secrecy constraint, only relative values may be used unless this initial value is known.

Each positional value is generated from the *binary complement of the preimage* of its larger neighbour, in a recursive cascade starting with the unary digit. This protects the information needed to derive the current state, and thus prevents leaking all of the values in that range.

Algorithm 1 Spiral Ratchet Initialization

Require: $(n, base, width) \in \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$
1: $seed \leftarrow random(0 \dots 2^{width})$
2: $state \leftarrow \{base = base, unary = hash(seed), pos = []\}$
3: **for** $i \leftarrow n - 1$ **to** 0 **do** ▷ Descending to associate index with degree
4: $seed \leftarrow hash(\sim seed)$ ▷ Secretly derive from the larger value
5: $\delta \leftarrow random(0 \dots base - 1)$
6: $state[pos][i] \leftarrow \{count = \delta, value = hash^{\delta+1}(seed)\}$
7: **end for**
8: **return** *state*

The number of digits is a balance between granularity and jump control. If the number of digits is large, increment on the unary element or higher digits will have require multiple hash operations in each zero cascade.

In practice, a 3-component (a unary spine and two positional digits) SHA3-256 spiral ratchet on base-256 has performed well. The maximum leap is 256^2, which means that one can calculate the state for $i + 256^3$ in the same time as $i + 256$.

5.2 Basic Operations

Basic operations on the spiral ratchet follow from the basic rules set out during initialization. It can be incremented sequentially, or leap to the next "zero" of any digit. Any interval can be efficiently found by combining increments and leaps.

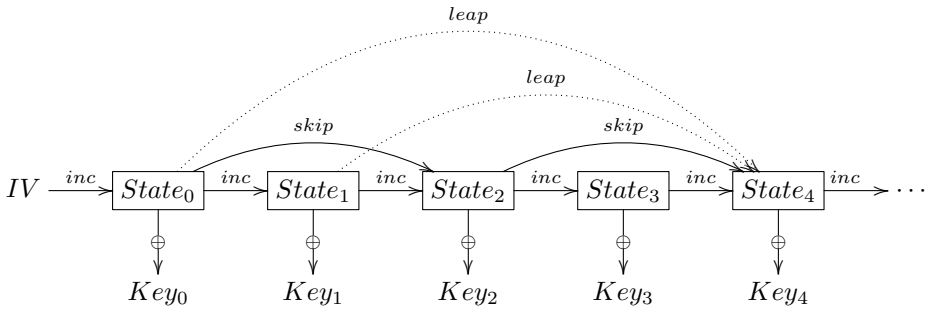

Figure 8: Spiral Ratchet Operations

Being a ratchet, it can never be "unwound" (there is no inverse or subtractive analogue). Access to earlier keys requires knowing an even earlier state, and moving forwards to the desired state from there. This is important for break-in resistance and backwards secrecy (e.g sharing a document from a point in time but no earlier).

5.2.1 Key Derivation

Generating output key material (OKM) from the ratchet state can be done in a myriad of ways. This paper will use the bitwise XOR function, as it is straightforward and efficient.

Algorithm 2 Generating a Key

```
1: function toKey(ratchet)
2:     sum ← ratchet[unary]
3:     for digit ∈ ratchet[pos] do
4:         sum ← acc ⊕ digit[value]
5:     end for
6:     sum
7: end function
```

5.2.2 Incrementing

Just like positional numerals, when a digit reaches its maximum, the next larger digit is incremented, and all lower digits are zeroed out. The zero cascade works similarly for spiral ratchets, but never repeats a value (i.e. the spiral property). Exactly like the initialization, each zero is derived from the preimage of the higher digit (Figure 9).

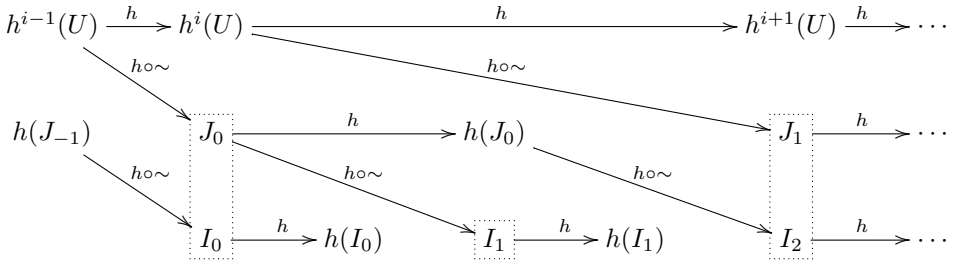

Figure 9: Zero Cascade for a Binary Spiral Ratchet

The special case of incrementing by one is the simplest way to demonstrate this (Algorithm 3). This functions by checking the lowest digit's counter, and if it's saturated, mark a carry and move to the next digit and recurse, bounded by the unary digit. Any carried digits are then re-zeroed by the preimage of their larger neighbour.

5.2.3 Arbitrary Jumps

Jumps by an arbitrary interval require a little more calculation (Figure 4). The interval is componentized, and increments each digit in turn. Each digit has an upper bound, so any remainder saved for the final step, where any zeroed digits are incremented to match the final count.

Each recursive zeroing is represented with a different value, both between digits and per-digit.

6 Security Considerations

Algorithm 3 Incrementing a Spiral Ratchet

```
1:  function INC(ratchet)
2:      {base, unary, pos} ← ratchet
3:      seed ← unary
4:      counter ← length(pos)
5:
6:      for i ← 0 to length(pos) − 1 do
7:          {count, value} ← pos[i]
8:          if count < base − 1 then                    ▷ Position not saturated
9:              seed ← value
10:             counter ← i
11:             ratchet[pos][i] ← {count = count + 1, value = hash(value)}
12:             break
13:         end if
14:     end for
15:
16:     if counter = length(pos) then    ▷ All positional values were saturated
17:         ratchet[unary] ← hash(unary)
18:     end if
19:
20:     if counter > 0 then
21:         for j ← counter − 1 to 0 do                  ▷ N.B. Descending
22:             seed′ ← pos[j][value]
23:             ratchet[pos][j][count] ← {count = 0, value = hash(∼ seed)}
24:             seed ← seed′
25:         end for
26:     end if
27:
28:     ratchet
29: end function
```

Algorithm 4 Spiral Ratchet Arbitrary Jump

```
1: function JUMP(ratchet, amount)
2:     {base, unary, pos} ← ratchet
3:     remaining ← amount
4:     seed ← unary
5:     carry ← []
6:     n ← length(pos)
7:
8:     for i ← 0 to n − 1 do
9:         if remaining = 0 then
10:            break
11:        end if
12:
13:        component_i ← remaining mod base^i
14:        remaining ← remaining − component_i
15:        δ_i ← component ÷ base^i
16:
17:        {count, value} ← pos[i]
18:        headroom ← base − count − 1
19:
20:        if δ_i > headroom then
21:            carry[i] ← steps − headroom
22:        else
23:            ratchet[pos][i][count] ← count + δ_i
24:            if remaining = 0 then
25:                seed ← hash^(δ_i − 1)(value)
26:                ratchet[pos][i][value] ← hash(seed)
27:            else
28:                carry[i] ← 0
29:            end if
30:        end if
31:    end for
32:
33:    if remaining > 0 then
34:        component_u ← remaining mod base^n
35:        δ_u ← component_u ÷ base^n
36:        ratchet[unary] ← hash^(δ_u)(unary)
37:    end if
38:
39:    for j ← length(acc) − 1 to 0 do                ▷ N.B. Descending
40:        seed ← hash^(acc[j] − 1)(∼ seed)
41:        ratchet[pos][j][value] ← hash(seed)
42:    end for
43:
44:    ratchet
45: end function
```

7 Acknowledgements

The spiral ratchet algorithm was developed at Fission Codes to support work on the WebNative File System (WNFS). Many thanks to the Fission Codes team for providing the freedom to do this work.

Thanks to Philipp Krüger at Fission Codes for feedback on the design, for writing the bulk of the TypeScript implementation, and for his patience as various parts of the algorithm were tweaked.

Thanks to Brendan O'Brien at Qri for the Go implementation, and for many discussions on the best way to explain the concept.

Thanks to Daniel Holmgren at **NOT YET PUBLIC** for his feedback on early versions of this algorithm.

Thanks to Juan Benet at Protocol Labs for encouraging me to write a more full-fledged description of the spiral ratchet than what was available in the Fission Codes online documentation.

The term "ratchet" for forward-secure key updating was introduced by Adam Langley in Pond [18]