Spiral Ratchet

A Recursive Hash System

Brooklyn Zelenka
Fission Codes
brooklyn@fission.codes

December 30, 2021
v1.0.0

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[2]), deterministically generating unique names, key derivation functions (KDFs), and so on. This paper introduces 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.

An increasing number of applications in open networks have no fixed topology of peers, and need to work in presence of a high latency ratio or network partitions. They require zero-interaction key agreement on many items, the ability to share a single item (and no others), a range of items, or elements from a point onwards. As such, there is a need to keep the historical information of the 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 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 in order to access the latest update.

The Spiral Ratchet improves on this situation. 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 or participants. It is configurable to permit jumps of any size and granularity.

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 number[7]. 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-encoding[9]. 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.

$$246_{16} = 2\times16^2 + 4\times16^1 + 6\times16^0 = 582_{10}$$

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 state 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 deterministic skip lists[3] (Figure 4) (but with each step pointing to a monotonically increasing value, rather than previous values).

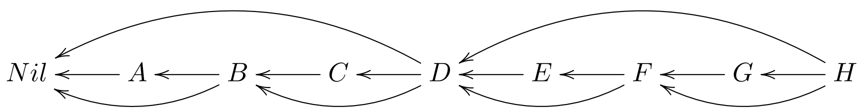

Figure 4: Skip List

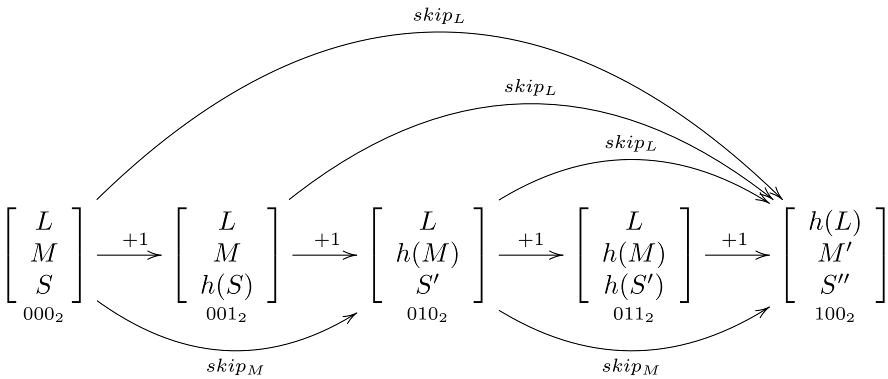

Figure 5: Simplified Compound Ratchet

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 (Figure 6): one component returns to its initial value, but the other is always moving forward. The motion is predictable and roughly circular, and never crosses itself. Extending the metaphor further: following the spiral in single steps is smooth, but leaps are possible by crossing to adjacent rings.

This is similar to a SkipNet[1], but rooted in a linear hierarchy (Figure 9) rather than a binary tree of rings. Also unlike rings, the Spiral Ratchet never repeats a value despite the counter being kept to a limited number of elements. The We call this the "spiral property."

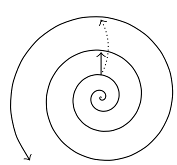

Figure 6: Spiral With Leaps

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.

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

Figure 7: Spiral Ratchet State

Note that Figure 7 is given 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: $(n, base, width) \in \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$

$seed \leftarrow random(0 \ldots 2^{width})$

$state \leftarrow \{base = base, unary = hash(seed), pos = []\}$

for $i \leftarrow n - 1$ **to** 0 **do** ▷ Descending to associate index with degree

    $seed \leftarrow hash(\sim seed)$ ▷ Secretly derive from the larger value

    $\delta \leftarrow random(0 \ldots base - 1)$

    $state[pos][i] \leftarrow \{count = \delta, value = hash^{\delta+1}(seed)\}$

end for

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 the calculation of the state $i + 256^3$ is approximately the same amount of work 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.

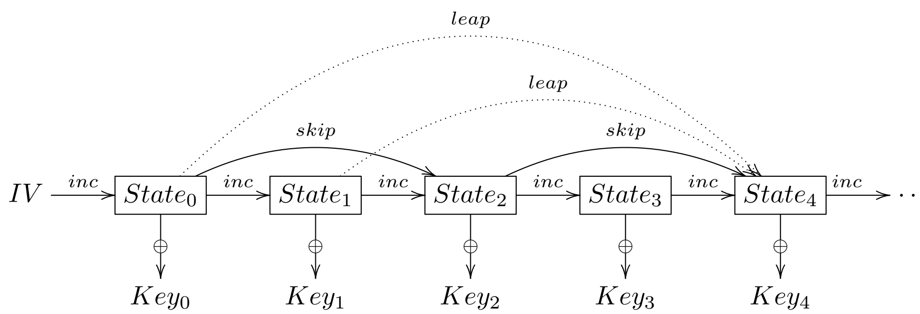

Figure 8: Spiral Ratchet Operations

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

```
function toKey(ratchet)
    sum ← ratchet[unary]
    for digit ∈ ratchet[pos] do
        sum ← acc ⊕ digit[value]
    end for
    sum
end function
```

5.2.2 Incrementing

As with 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).

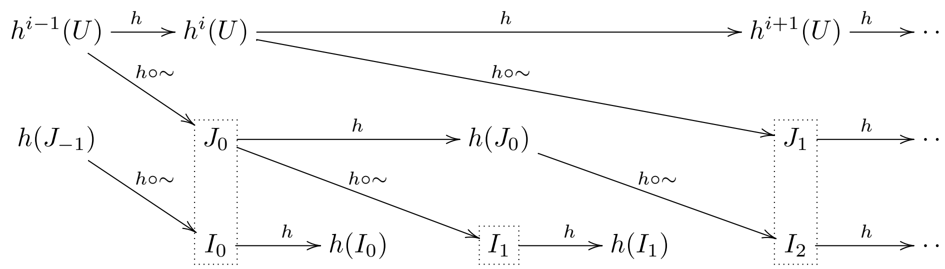

Figure 9: Zero Cascade for a Binary Spiral Ratchet

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 (Algorithm 4). The interval δ is componentized, and each digit incremented in ascending order. Every positional digit has an upper bound, so any remainder saved for the final step, where any zeroed digits are incremented to match the final count. Arbitrary jumps run in $\mathcal{O}(\log_b \delta)$ when $\delta < b^{bn}$ (the normal operating interval that the ratchet is tuned for). As the unary hash chain dominates as δ becomes very large, the complexity changes to $\mathcal{O}(\frac{\delta}{b^n})$.

6 Security

Encryption is a form of direct access control. Much like how a bearer token may grant access to some resource, knowing a key grants access to the cleartext. There is no in-built way to revoke this access once the key is compromised. Updates rely on forward secrecy for protection, and key rotation in the case where a breach becomes known. Algorithms like the Double Ratchet[8] have a self-healing property where there key is automatically rotated as a matter of course. This necessarily requires trusting the other party.

The Spiral Ratchet trades off self-healing for permissionless key agreement between an unbounded and growable set of peers. If the secured communications are between exactly two parties, the Double Ratchet is typically a better choice.

There are two levels of security for a KDF: the external key generation, and its internal state. The internal state is itself based on hierarchical hash chains to assure backwards secrecy even for those that have been given access to the

Algorithm 3 Incrementing a Spiral Ratchet

```
function INC(ratchet)
    {base, unary, pos} ← ratchet
    seed ← unary
    counter ← length(pos)

    for i ← 0 to length(pos) − 1 do
        {count, value} ← pos[i]
        if count < base − 1 then                          ▷ Position not saturated
            seed ← value
            counter ← i
            ratchet[pos][i] ← {count = count + 1, value = hash(value)}
            break
        end if
    end for

    if counter = length(pos) then        ▷ All positional values were saturated
        ratchet[unary] ← hash(unary)
    end if

    if counter > 0 then
        for j ← counter − 1 to 0 do                          ▷ N.B. Descending
            seed′ ← pos[j][value]
            ratchet[pos][j][count] ← {count = 0, value = hash(∼ seed)}
            seed ← seed′
        end for
    end if

    ratchet
end function
```

Algorithm 4 Spiral Ratchet Arbitrary Jump

```
function JUMP(ratchet, amount)
    {base, unary, pos} ← ratchet
    remaining ← amount
    seed ← unary
    carry ← []
    n ← length(pos)

    for i ← 0 to n − 1 do
        if remaining = 0 then
            break
        end if

        component_i ← remaining mod base^i
        remaining ← remaining − component_i
        δ_i ← component ÷ base^i

        {count, value} ← pos[i]
        headroom ← base − count − 1

        if δ_i > headroom then
            carry[i] ← steps − headroom
        else
            ratchet[pos][i][count] ← count + δ_i
            if remaining = 0 then
                seed ← hash^(δ_i − 1)(value)
                ratchet[pos][i][value] ← hash(seed)
            else
                carry[i] ← 0
            end if
        end if
    end for

    if remaining > 0 then
        component_u ← remaining mod base^n
        δ_u ← component_u ÷ base^n
        ratchet[unary] ← hash^(δ_u)(unary)
    end if

    for j ← length(acc) − 1 to 0 do                ▷ N.B. Descending
        seed ← hash^(acc[j] − 1)(∼ seed)
        ratchet[pos][j][value] ← hash(seed)
    end for

    ratchet
end function
```

KDF. Not only is the previous state not calculable, other metadata such as the number of steps is also hidden, even when the internal state is known.

This is a flexible setup: revealing one OKM provides access to a single version. Providing the XOR of the unary digit and a nonzero number of positional elements gives (self-healing) access to a limited interval of updates. Access to the entire internal state grants access from a point onwards, including potentially the initial state.

7 Conclusion

This paper has presented the Spiral Ratchet, an algorithm for key agreement on data structures with large numbers of updates and unbounded numbers of participants. Arbitrary search forward from a point can be done in $\mathcal{O}(\log_b \delta)$ time in the expected context, backwards access is always impossible, and counts cannot be inferred from the internal state.

8 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 develop this algorithm.

Thanks to Philipp Krüger at Fission Codes for feedback on the design, for writing the bulk of the TypeScript implementation[4], and for his patience as various parts of the algorithm were tweaked while that code was already inflight.

Thanks to Brendan O'Brien at Qri for the Go implementation[6], 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 fuller description of the Spiral Ratchet than what was available in the WebNative online documentation.

The term "ratchet" for forward-secure key updating was introduced by Adam Langley in Pond[5]

References

[1] Nicholas J.A. Harvey et al. *SkipNet: A Scalable Overlay Network with Practical Locality Properties*. 2002. URL: `https://www.ic.unicamp.br/~celio/peer2peer/skip-net-graph/skipnet-tr-2002-92.pdf`. (accessed: 2021-12-26).

[2] Neil M. Haller. *The S/KEY One-Time Password System*. RFC 1760. Feb. 1995. DOI: `10.17487/RFC1760`. URL: `https://rfc-editor.org/rfc/rfc1760.txt`.

[3] Thomas Papadakis J. Ian Munro and Robert Sedgewick. "Deterministic Skip Lists". In: *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1992, pp. 367–375. URL: `https://www.ic.unicamp.br/~celio/peer2peer/skip-net-graph/deterministic-skip-lists-munro.pdf`. (accessed: 2021-12-26).

[4] Philipp Krüger. *WebNative Spiral Ratchet*. Fission Codes. 2021. URL: `https://github.com/fission-suite/webnative/blob/e1eed4e750c668a6f54d9be701dc17e286b9eff4/src/fs/data/private/spiralratchet.ts`.

[5] Adam Langley. *Pond README*. 2012. URL: `https://github.com/agl/pond/commit/7bb06244b9aa121d367a6d556867992d1481f0c8`.

[6] Brendan O'Brien. *go-wnfs Spiral Ratchet*. Qri. 2021. URL: `https://github.com/qri-io/wnfs-go/blob/62915924c866f7106aecacccd14e57bd1b62850f/private/ratchet/ratchet.go`.

[7] *Peano axioms §Arithmatic*. Wikipedia. URL: `https://en.wikipedia.org/wiki/Peano_axioms#Arithmetic`. (accessed: 2021-12-26).

[8] Trevor Perrin and Moxie Marlinspike. Version Revision 1, 2016-11-20. Signal Foundation. Nov. 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[9] Benjamin C. Pierce. "Types and Programming Languages". In: The MIT Press, 2002. Chap. 5.2, p. 60. ISBN: 978-0-262-16209-8.