Skip Ratchet

A Recursive Hash System

Brooklyn Zelenka
Fission Codes
brooklyn@fission.codes

January 4, 2022
v1.0.0

Abstract

Hash chains are a simple way to deterministically generate key material, but they are inefficient in many situations, such as synchronizing the heads of a forward secret stream. This paper presents the Skip Ratchet, a novel symmetric key derivation function 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"[5]) for pre-computation resistance, forward secrecy (FS), one-time schemes (e.g. S/KEY[3]), deterministically generating unique names, key derivation functions (KDFs), and so on. This paper introduces the Skip Ratchet (SR), a hierarchical and iterative hashing system capable of efficiently making large leaps in hash count, while preserving forward secrecy in a passive setting.

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. The lack of knowledge of any information about the peers implies the use of passive security methods based directly on data.

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 also a need to keep the historical information of the internal state secret.

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 Skip Ratchet improves on this situation. It is able to synchronize in sublinear time, supports efficient arbitrary access in the forward direction, its internal state is backwards secret, and does not leak metadata such as the number of updates or participants to anyone. It is configurable to permit jumps of any size and granularity.

3 Numeral Intuition

The problem of how to efficiently calculate the n^{th} step of a ratchet in less than n steps is structurally similar to how we efficiently represent large numbers. There are many systems for doing so. The two most common are tally marks (unary) and positional numeral systems. The Spiral Ratchet uses both of these concepts.

3.1 Unary Hashing

Unary is one of the simplest forms of expressing 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[9]. These numerals are represented by a zero element 0, and successors S (Figure 1). This is a very straightforward method, but is neither space nor time efficient.

Peano numbers may expressed as functions via Church-encoding[11]. 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 $h(h(h(I)))$ (Figure 1).

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

$$Peano(3) \Rightarrow S(S(S(0)))$$
$$h_{chain}(3) \Rightarrow h(h(h(I))) \equiv h^3(I)$$

Figure 1: Peano Number $\leftrightarrow$ Hash Chain

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 2: Componentized Hexadecimal

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

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

Figure 3: Compound Hash

Unlike a positional number system used in arithmetic, a KDF "forgets" its internal state, combining elements and flattening its structure. There are many methods of combination, including further hashing; this paper will use bitwise XOR (Figure 4).

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 consistent bases across all positional digits.

4 Bounding

Unlike number systems for arithmetic, the Skip Ratchet has a fixed number of "digits" in its internal state. This limits the jumps to fixed intervals, with a maximum jump interval. It is possible to create variants that deterministically generate increasingly large digits as needed, but this would leak data about the range that the count is currently in. This indistinguishability property is desirable in many use cases, such as for securing updatable documents on an public network.

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

Figure 4: Compound Hash Key Derivation

While advancing the ratchet 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, and therefore arbitrary jumps require a carry (Algorithm 4). This is structurally similar to deterministic skip lists[7] (Figure 5), but with each step pointing to a monotonically increasing value, rather than previous ones.[1]

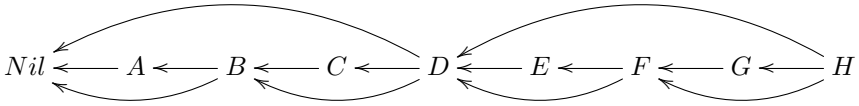

Figure 5: Skip List

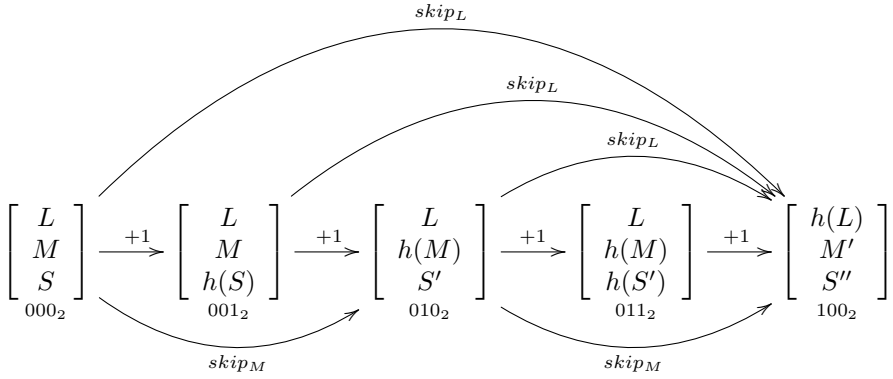

Figure 6: Simplified Skip Ratchet

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

[1] While the analogy to fingered DAGs such as skip lists in the name is apt, the SR is distinct from other DAG-structured hash constructions like the Merkle Tree[6] or Hash Calendar[2]. A Skip Ratchet does not aggregate digests into a signature, but rather counts by iteratively hashing the same value repeatedly, in tiers.

is called an "epoch" (e.g. "the 300s epoch" or "the $h^{42}(J)$ epoch").[2]

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 extends outwards rather than crossing itself. Extending the metaphor further: following the spiral in single steps is smooth, but leaps are possible by jumping to adjacent rings (Figure 7). Unlike the repeating numerals found in most counting systems, the Skip Ratchet never repeats a value despite the counter being kept to a limited number of elements. We call this the "spiral property.".

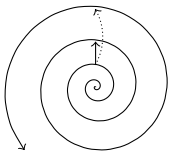

Figure 7: Spiral With Leaps

5 Skip Ratchet

The Spiral Ratchet's state 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.

$$S : ((Count_0, Value_0), (Count_1, Value_1) \dots (Count_{n-1}, Value_{n-1}), U)_b$$

Figure 8: Skip Ratchet State

Note that Figure 8 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

A Skip Ratchet is deterministically derived from a hash function h, a numeric base b, a positional digit count n, and an initialization vector IV, generating a *seed state*. We distinguish between configurations by referring to their (n, b, h) triple, such as $(3, 256, \mathsf{SHA3})$ Skip Ratchet.

$$(n, b, h, IV) \rightarrow S$$

Figure 9: Seed Generation

[2]The term "epoch" is used here in the generic or Ethereum[12] sense, and is a slightly different from some other ratchet constructions such as the Double Ratchet[1].

To prevent leaking the iteration count, all of the positional values in the seed state are immediately incremented by a random value. This randomized origin is treated as the ratchet's starting state. Given the backward 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 (Algorithm 1). This protects the information needed to derive the current state, and thus prevents leaking all of the values in that range.

Algorithm 1 Skip Ratchet Initialization

Require: $(n, b, \lambda) \in \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$

1: $seed \xleftarrow{\$} \{0,1\}^\lambda$
2: $pos \coloneqq []$
3: $unary \coloneqq hash(seed)$
4: **for** $i \leftarrow (n-1) \ldots 0$ **do** ▷ Descending to associate index with degree
5: $\quad \delta \xleftarrow{\$} 0 \ldots b-1$
6: $\quad seed \coloneqq hash(\sim seed)$ ▷ Secretly derive from the larger value
7: $\quad pos[i].count \coloneqq \delta$
8: $\quad pos[i].value \coloneqq hash^{\delta+1}(seed)$
9: **end for**
10: **return** $b, unary, pos$

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 (Figure 11).

In practice, a 3-degree[3] base-256 SHA3 Skip Ratchet has performed well. The maximum leap is 256^2, so the state $i + 256^3$ is (approximately) the same amount of work as $i + 256$. Given the small constant factor of hardware-accelerated hashing and the monotonicity of a ratchet, relatively large bases are often viable.

5.2 Basic Operations

Basic operations on the Skip Ratchet follow from the 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 skips (Figure 10).

It is sometimes helpful to distinguish between a small and large skip, and so the terms "skip" and "leap" may be used for this purpose, despite them functioning identically but on different positions. The term "increment" (or "inc") is only used for the special case of the smallest movement (by one).

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

[3]Unary spine and two positional digits

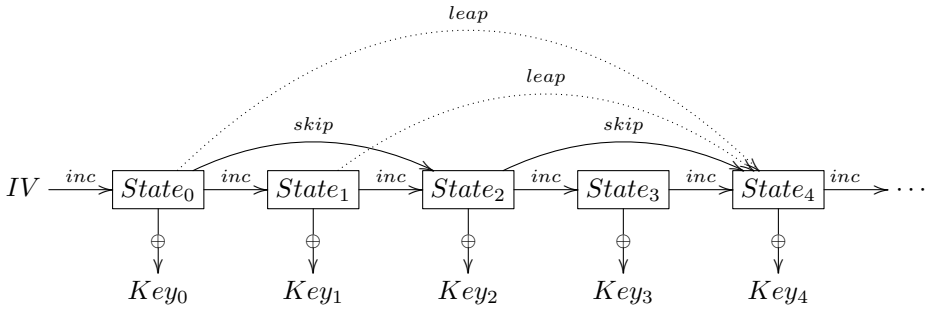

Figure 10: Binary Skip Ratchet Operations

moving forward 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:     return sum
7: 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 Skip 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 11).

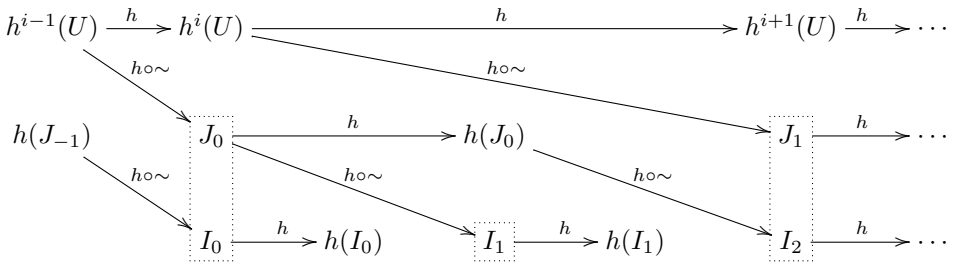

Figure 11: Zero Cascade for a Binary Skip Ratchet

The special case of incrementing by one is the simplest way to demonstrate this (Algorithm 3). It 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 $\delta < b^{b(n+1)}$ (the normal operating interval that the SR 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 inbuilt 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[10] have post-compromise security, where there key is automatically rotated as a matter of course. This necessarily requires trusting the other party.

The Skip Ratchet has two security domains: the external key generation, and its internal state. The internal state is itself based on hierarchical hash chains to assure forward secrecy, even for those that have been given access to the KDF's internal state. 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.

A flexible amount of information may be shared with limited, passive post-compromise security. Revealing one OKM provides access to the cleartexts encrypted with that key (e.g. a single version of one file). Providing the XOR of

Algorithm 3 Incrementing a Skip Ratchet

```
 1: function INC(ratchet)
 2:     {base, unary, pos} ← ratchet
 3:     seed ≔ unary
 4:     counter ≔ length(pos)
 5:
 6:     for i ← 0 . . . 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
12:             ratchet.pos[i].value ≔ hash(value)}
13:             break
14:         end if
15:     end for
16:
17:     if counter ≟ length(pos) then    ▷ All positional values were saturated
18:         ratchet[unary] ≔ hash(unary)
19:     end if
20:
21:     if counter ≟ 0 then
22:         for j ← (counter − 1) . . . 0 do                ▷ N.B. Descending
23:             seed′ ≔ pos[j].value
24:             ratchet[pos][j].count ≔ 0
25:             ratchet[pos][j].value ≔ hash(∼ seed)
26:             seed ≔ seed′
27:         end for
28:     end if
29:
30:     ratchet
31: end function
```

Algorithm 4 Skip Ratchet Arbitrary Jump

function JUMP($ratchet, amount$)
  $\{base, unary, pos\} \leftarrow ratchet$
  $remaining \coloneqq amount$
  $seed \coloneqq unary$
  $carry \coloneqq []$
  $n \coloneqq length(pos)$

  for $i \leftarrow 0 \ldots n-1$ **do**
    if $remaining \overset{?}{=} 0$ **then**
      break
    end if

    $component_i \coloneqq remaining \mod base^i$
    $remaining \coloneqq remaining - component_i$
    $\delta_i \coloneqq \frac{component}{base^i}$ ▷ No remainder because ascending in steps

    $\{count, value\} \leftarrow pos[i]$
    $headroom \coloneqq base - count - 1$

    if $\delta_i \overset{?}{>} headroom$ **then**
      $carry[i] \coloneqq steps - headroom$
    else
      $ratchet.pos[i].count \coloneqq count + \delta_i$
      if $remaining \overset{?}{=} 0$ **then**
        $seed \coloneqq hash^{\delta_i - 1}(value)$
        $ratchet.pos[i].value \coloneqq hash(seed)$
      else
        $carry[i] \coloneqq 0$
      end if
    end if
  end for

  if $remaining \overset{?}{>} 0$ **then**
    $component_u \coloneqq remaining \mod base^n$
    $\delta_u \coloneqq \frac{component_u}{base^n}$
    $ratchet.unary \coloneqq hash^{\delta_u}(unary)$
  end if

  for $j \leftarrow (length(carry) - 1) \ldots 0$ **do** ▷ N.B. Descending
    $seed \coloneqq hash^{carry[j]-1}(\sim seed)$
    $ratchet.pos[j].value \coloneqq hash(seed)$
  end for

  return $ratchet$
end function

the unary digit and a nonzero number of positional elements gives (self-healing) access to the remaining number of updates in that epoch. Access to the entire internal state grants access from a point onwards, including potentially the initial state.

The Skip Ratchet trades off post-compromise 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 due to its post-compromise security. Such a model could be added on top of a Skip Ratchet by placing it inside a Double Ratchet, distributing a freshly initialized SR to a list of trusted peers on every i steps, or any other method of rotating the unary digit.

7 Conclusion

This paper has presented the Skip Ratchet, an algorithm for key agreement for streams of encrypted data with unbounded and changing participants. Arbitrary forward search can be performed 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 Skip 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 in-flight.

Thanks to Brendan O'Brien at Qri for the Go implementation[8], and for many discussions on the best way to explain the concept.

Many thanks to Quinn Wilton for her feedback on various drafts of this paper, suggestions, and insights from the broader literature.

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 Skip Ratchet than what was available in the WebNative online documentation.

References

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. *The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol*. Feb. 2020. URL: https://eprint.iacr.org/2018/1037.pdf.

[2] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. *Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees*. URL: https://guardtime.com/files/BuKL13.pdf. (accessed: 2022-01-03).

[3] 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.

[4] Philipp Krüger. *WebNative Spiral Ratchet*. Fission Codes. 2021. URL: https://github.com/fission-suite/webnative/blob/e1eed4e750c668a6f54d9be701dc17e286b9ef1
src/fs/data/private/spiralratchet.ts.

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

[6] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*. CRYPTO '87. Berlin, Heidelberg: Springer-Verlag, 1987, pp. 369–378. ISBN: 3540187960.

[7] J. Ian Munro, Thomas Papadakis, 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).

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

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

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

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

[12] Gavin Wood. *Ethereum: A Secure Decentralized Generalized Transaction Ledger*. Version Berlin (fabef25). Ethereum Foundation. Dec. 2021. URL: https://ethereum.github.io/yellowpaper/paper.pdf. (accessed: 2022-01-03).