A Cryptanalysis of IOTA's `Curl` Hash Function

Michael Colavita
Harvard College

Garrett Tanzer
Harvard College

May 27, 2018

> *"Don't roll your own crypto" is a compulsory uttered mantra that serves as a good guiding principle for 99.9% of projects, but there are exceptions to the rule.*[1]
>
> - David Sønstebø, Founder of IOTA

Abstract

In this paper, we analyze the security of IOTA's now-deprecated cryptographic hash function `Curl`. We independently reproduce and formalize the results of Heilman et al. [7] by demonstrating a second-preimage attack, a related digest attack, and a constructive full-state collision. We conclude that `Curl` is not a secure cryptographic hash function.

1 Introduction

The cryptocurrency landscape is defined by agile development: in such a volatile speculative market, developers are strongly incentivized to play fast and loose with their schemes' security, publishing informal arguments based on heuristics in whitepapers as they let rigorous security proofs fall by the wayside. This attitude has led to a spectrum of compromises, ranging from IOTA, where fundamental cryptographic primitives were replaced with proprietary, unvetted solutions, to Ethereum, where a cryptographic backbone lends a false sense of security to critical smart contracts [12].

The former of these cases is remarkable for its notoriety. Until it was replaced in early August by `Keccak` [13], the cryptographic hash function `Curl` was a vital component of signatures in IOTA—at the time the the world's 9th largest cryptocurrency with a market cap of over $1.1 billion [4]. In September, Heilman et al.'s vulnerability report was published, revealing critical flaws in `Curl`'s collision resistance property. In response, the IOTA team claimed because its "tangle" blockchain variant makes use of a centralized Coordinator to validate transactions, the security of the scheme relies only on `Curl`'s one-wayness—but of course there is no security proof for this, and much of the network's central functionality is undocumented or closed-source. In February, the waters were muddied even further when the private email chain preceding the disclosure was leaked, revealing more conflicting claims between the two parties and effecting skepticism from cryptocurrency enthusiasts of the report's conclusions [9].

In this paper, we reproduce and formalize the results from Heilman et al.'s disclosure report, providing incontrovertible evidence that `Curl` is not collision-resistant. In Section 2, we provide background on the typical guarantees expected from cryptographic hash functions and the techniques—in particular, differential cryptanalysis—used to break them. In Section 3, we describe `Curl` as an instantiation of the sponge construction and analyze its mixing properties. In Section 4, we formalize several attacks on `Curl`'s cryptographic properties, including a constructive full-state collision attack that allows us to generate unbounded numbers of fixed-length colliding pairs. In Section 5, we evaluate the relevance of these attacks to `Curl` at a structural level, rather than to the constants and parameters of its current instantiation, and in Section 7 we conclude. Our multithreaded C implementation of the central collision attack is publicly available at curl-collisions.

[1] We argue that IOTA does not lie in this 0.1%.

2 Background

A cryptographic hash function is defined as an efficiently computable function $h : \{0,1\}^* \rightarrow \{0,1\}^k$ [10] such that for all probabilistic polynomial time adversaries A with access to h, the following informal properties hold [3]:

> **Pre-Image Resistance:** Given a random y in the range of h, A outputs an x such that $h(x) = y$ with negligible probability in k.
>
> **Second Pre-Image Resistance:** Given a random x, A outputs an $x' \neq x$ such that $h(x) = h(x')$ with negligible probability in k.
>
> **Collision Resistance:** A outputs an x, x' such that $h(x) = h(x')$ with negligible probability in k.

We call the input to the hash function the *message* or the *plaintext*, and the output of the function the *digest* or the *hash*. Because the existence of cryptographic hash functions implies $\mathsf{P} \neq \mathsf{NP}$, for practical constructions the above properties are assumed only after years of vetting by the academic community and experienced cryptanalysts. One fundamental technique used to discover vulnerabilities in practical cryptographic primitives is differential cryptanalysis.

Differential cryptanalysis was first discovered secretly in 1974 by the Data Encryption Standard (DES) design team at IBM, who used foreknowledge of the attack to harden the algorithm's S-boxes—small nonlinear functions usually implemented as a lookup table. A decade later, Biham and Shamir independently rediscovered and applied the attack to DES, noting its remarkable resilience to the approach [2]. The term broadly refers to techniques, usually chosen plaintext attacks, that use statistical relationships between *differences* in plain- and ciphertexts, which are usually defined with XOR as $\Delta = x \oplus y$. These pairs of differences, or *differentials*, between the input and output of an S-box are used to trace dependencies as they propagate across multiple iterations of a round function. The resulting path, called a *differential characteristic*, describes a $\Delta_Y = H(X \oplus \Delta_X) \oplus H(X)$ that occurs with significant probability. While this analysis is usually used to extract secret keys from block ciphers, it can also be used to find collisions in a hash function when $\Delta_Y = 0$, or to perform related digest attacks for other values of Δ_Y [14].

3 Description of `Curl`

`Curl` is based on the sponge construction paradigm pioneered by the `Keccak/SHA-3` cryptographic hash function [13]. The traditional sponge construction is defined as follows [1]:

Let f be a permutation on b bits. The sponge construction has a state, or "sponge" $s \in \{0,1\}^b$, usually initialized to $s_0 = 0^b$. We can define this state as the concatenation of two substrings $s = r \| c$ at all points in time. The input message M is padded with the function $\mathtt{pad}_{|r|}$ and split into $|r|$-bit blocks. Then to compute the hash, we proceed with two phases:

- **Absorbing:** For each of $\lceil \frac{|M|}{|r|} \rceil$ input blocks m_i of length $|r|$, sequentially set $r \leftarrow r \oplus m_i$ then $s \leftarrow f(s)$. By s_i, r_i, and c_i we denote the state of s, r, and c after absorbing m_i.
- **Squeezing:** Starting with the empty string $\varnothing$, while the output is less than the desired length, append r to the cumulative output and set $s \leftarrow f(s)$. Finally, truncate any extra bits so that the length of the output is exactly the desired length, rather than a multiple of $|r|$.

`Curl` differs from the standard construction in two significant ways:

1. f is a permutation on t trits, rather than b bits. That is to say, $s \in \{-1,0,1\}^t$ and $s_0 = 0^t$. To the best of our knowledge, this has no effect on the properties of the hash function except that it harms real-world performance.
2. `Curl` uses no padding function. Instead, we replace the absorbing phase's update rule $r \leftarrow r \oplus m_i$ (or the analogue, addition in $\mathbb{F}_3$) with the rule $r \leftarrow m_i$. If $|m_i| < r$ at the end of the input, set only the first $|m_i|$ trits of r to $|m_i|$, and leave the rest unchanged. This will be relevant to the Collision Resistance Attack we present in Section 4.4.

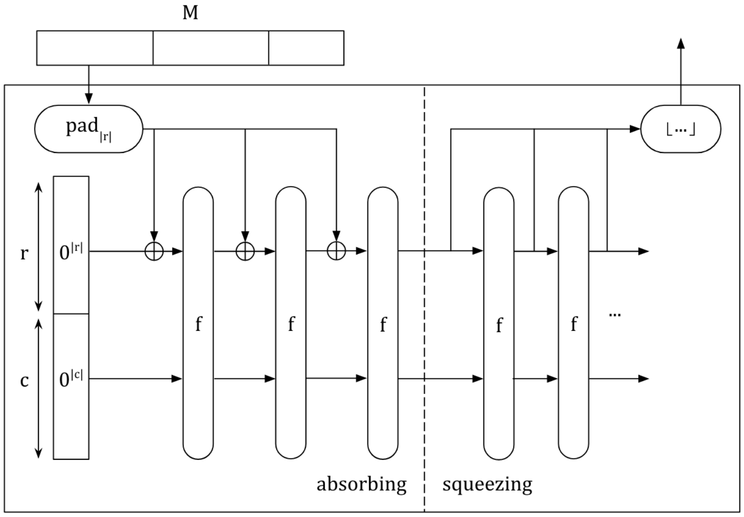

Figure 1: The traditional sponge construction [1].

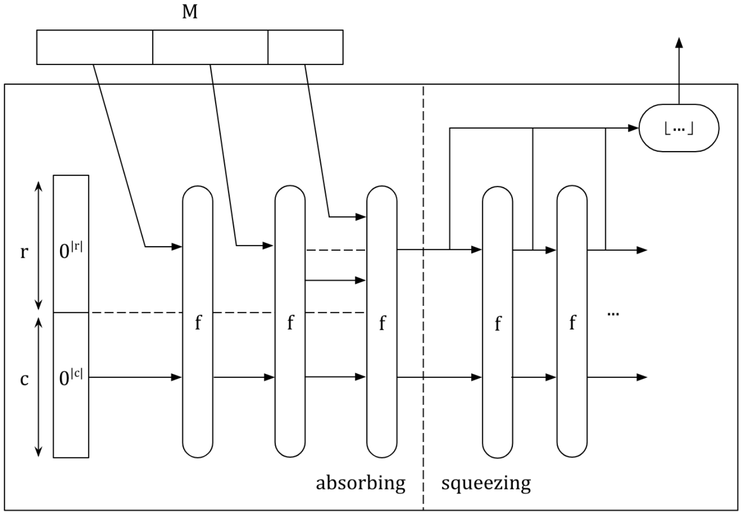

Figure 2: `Curl`'s sponge construction.

```
sbox = [[1, 1, -1],
        [0, -1, 1],
        [-1, 0, 0]]

def permute(x, y):
    return sbox[x+1][y+1]

def transform(state):
    for r in range(81):
        nstate = 729 * [0]
        t = 0
        p = 0
        for s in range(729):
            p = t
            t += 364 if t < 365 else -365
            x = state[p]
            y = state[t]
            nstate[s] = permute(x, y)
        state = nstate
    return state
```

Figure 4: Python implementation of `permute` and `transform`, for concreteness. `state` is an array of length 729.

The practical implementation of `Curl` is instantiated with the parameters $|r| = 243$ and $|c| = 486$, and implements f with 27 nested calls to the function `transform`. Mathematically, $\texttt{transform} : \{-1, 0, 1\}^{729} \to \{-1, 0, 1\}^{729}$ is defined in terms of the S-Box `permute` as follows:

$$\texttt{transform}(x)_i = \texttt{permute}(x_{364i \mod 729}, x_{364(i+1) \mod 729})$$

			y	
		-1	0	1
	-1	1	1	-1
x	0	0	-1	1
	1	-1	0	0

Figure 3: `Curl`'s S-box, `permute`.

3.1 Analysis of `transform`

We now show that `transform` for this specific choice of S-box is a permutation, satisfying the requirements of the sponge construction. We first show the following useful lemma.

Lemma 1 (Inversion Given a Single Input Trit)**.** *Given the output* $\texttt{transform}(x) = y$ *and a single input trit* x_i *for some position* i*,* x *can be computed efficiently if either* $\texttt{permute}(\cdot, k)$ *or* $\texttt{permute}(k, \cdot)$ *is bijective for all* k*.*

Proof. Suppose, for some choice of S-box, $\texttt{permute}(\cdot, k)$ is bijective for all k. Because the domain and range of $\texttt{permute}(\cdot, k)$ are of cardinality 3 for fixed k, we can easily precompute the j such that $\texttt{permute}(j, k) = \ell$ for every k and ℓ.

		ℓ		
		-1	0	1
	-1	1	0	-1
k	0	0	1	-1
	1	-1	1	0

Figure 5: The inversion table for `Curl`'s S-box.

Let $g_i = 364i \mod 729$. We have per the definition above that $\texttt{transform}(x)_i = \texttt{permute}(x_{g_i}, x_{g_{i+1}})$. Suppose without loss of generality that we are given $x_{g_{i+1}}$ as our single input trit. Per above, x_{g_i} is fully determined and we can use the known y_i and $x_{g_{i+1}}$ to compute this value. Next, using y_{i-1} and x_{g_i}, we can compute $x_{g_{i-1}}$. We repeat this procedure to compute x_{g_i} for all i. Because $\gcd(364, 729) = 1$, $\{g_i : 0 \leq i < 729\}$ contains all integers in $[0, 729)$. Therefore, this procedure computes x_i for all i, and each of these values is fully determined by the single known trit of x. Thus, by induction, this procedure produces a complete x such that $\texttt{transform}(x) = y$.

Suppose instead, for some choice of S-box, $\texttt{permute}(k, \cdot)$ is bijective for all k. We use the same process, proceeding forward instead of backward to fully determine the input string. Thus, in either case, we can easily invert given a single input trit.

Furthermore, consider the situation in which our provided $x_{g_{i+1}}$ is incorrect. By incorrect, we mean that there exists no x with the provided value $x_{g_{i+1}}$ at position g_{i+1} such that $\texttt{transform}(x) = y$. Per above, our inference would succeed as we compute the other 728 trits, as they are fully determined by our choice of $x_{g_{i+1}}$. However, if we repeat our inference procedure a 729th time, we will be inferring $x_{g_{i+1}}$ given y_{i+1} and our inferred $x_{g_{i+2}}$. Suppose we infer that our given value is correct. This contradicts our original assumption, as we now have that each of the output trits are produced from the inferred input trits, while we know that our inferred $x_{g_{i+1}}$ is incorrect. Thus, we must infer a value other than the known $x_{g_{i+1}}$. Therefore, we apply our inference for an additional step, and if our inferred $x_{g_{i+1}}$ contradicts the given one, we report that no such x exists. ■

Using Lemma 1, we can construct a simple $O(1)$-time algorithm to compute $\texttt{transform}^{-1}$. First, we note that `Curl`'s S-box satisfies the bijectivity condition given above; $\texttt{permute}(\cdot, k)$ is bijective for all k. Given the output $\texttt{transform}(x) = y$, we assume $x_0 := -1$ and use the inversion algorithm in Lemma 1 to compute $\texttt{transform}^{-1}(y)$. If the algorithm determines there is no inverse with $x_0 = -1$, we assume $x_0 := 0$ and repeat the process. If this also fails, we assume $x_0 := 1$ and repeat the process. If this fails, the output has no corresponding input. We can now proceed to prove our main theorem.

Theorem 1 (`transform` is a Permutation)**.** *For the choice of S-box used in `Curl`'s `permute` function, `transform` is a permutation.*

Proof. As the domain and range of `transform` are of the same cardinality, showing that for every y there exists exactly one x such that $\texttt{transform}(x) = y$ suffices to prove that `transform` is a permutation. Suppose, for the sake of contradiction, that there exist x, x' such that $x \neq x'$ and $\texttt{transform}(x) = \texttt{transform}(x') = y$. Note that $x_i \neq x'_i$ for every i, as otherwise Lemma 1 implies that $x = x'$ by using x_i as our known trit. Furthermore, if these x and x' exist, we can infer them from a single set of trits x_i, x'_i using the algorithm given above.

Thus `transform` is a permutation if and only if there do not exist trits $x_i \neq x'_i$ and an output $y : \{-1, 0, 1\}^{729}$ such that the algorithm above succeeds for both fixed input trits x_i and x'_i on output y. To show this is the case, we construct an inference graph, where each node represents a pair of trits (z_j, z'_j). Without loss of generality, we reorder the x and y such that inference proceeds in reverse sequential order. Thus, every edge from a given node corresponds to a trit in y and transitions to the state corresponding to the inferred (z_{j-1}, z'_{j-1}). Per the argument above, the algorithm yields success if and only after 729 transitions from our original state (x_i, x'_i) we arrive back at our original state. Therefore, we must show that this graph contains no cycles of length 729 to prove our theorem.

To simplify the graph, we note that there are no nodes corresponding to (z_j, z'_j) where $z_j = z'_j$, as this would imply $x_i = x'_i$ for some i. Thus we construct the following graph:

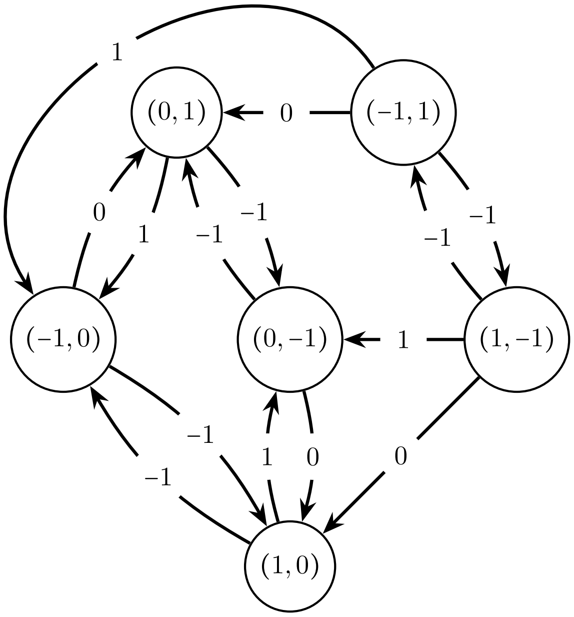

Figure 6: The inference graph for `Curl`'s S-box.

Using this inference graph, we must determine if there are cycles of length 729. We exponentiate the graph's adjacency matrix to the power of 729, yielding a matrix with all zeros on the diagonal. This implies there are no cycles of length 729 and proves that `transform` is a permutation.[2]

■

3.2 Differential Propagation

We note that if we modify a given trit in the state of the sponge, the `transform` function propagates this difference throughout the entire state. However, the rate at which it does so is relatively slow, which becomes vital for our later cryptanalysis. We prove the following useful lemmas.

Lemma 2 (Dependent States between Rounds)**.** *Consider the sponge state s_k. If there is a difference at position p, this difference will propagate to at most the positions $(727p \mod 729)$ and $(727p - 1 \mod 729)$ in s_{k+1}.*

Proof. The description of `Curl`'s `transform` function uses a set of indices $i_0, \dots, i_{728}$ generated by $i_j = 364j \mod 729$. To compute the value at position j in state s_{k+1}, we apply the S-box to the values $i_j = 364j \mod 729$ and $i_{j-1} = 364(j+1) \mod 729$ in the state s_k. We can invert this relationship to determine all states from s_{k-1} that affect position j in s_k. We note that 727 is the modular inverse of 364 modulo 729. Given that position j in state s_k is determined by positions $364j$ and $364(j-1)$ in state s_{k+1}, a given state p must satisfy either $p \equiv 364j \mod 729$ or $p \equiv 364(j-1) \mod 729$. Therefore $j \equiv 727p \mod 729$ or $j \equiv 727p - 1 \mod 729$. Therefore only the states $(727p \mod 729)$ and $(727p - 1 \mod 729)$ are affected. ■

Lemma 3 (Exponential Fan-out of State Differentials)**.** *Consider the intermediate states of two sponges after k rounds with only one differential at position p. After ℓ additional rounds, all trits that differ lie in a contiguous modular region of size at most 2^ℓ starting at position p'_ℓ, which we can solve for analytically.*

[2] Note, however, that this graph does contain cycles of length 2. Thus, this S-box does not yield a permutation for states of even size.

Proof. Given Lemma 2, we note that a single change can affect at most two positions in the next round. Given a differential at position p, the following round can only have differentials at positions $727p \mod 729$ and $727(p-1) \mod 729$. Continuing for another round, we now have potential differentials in positions:

$$\begin{aligned}
727(727p) &\mod 729 = 727^2 p \mod 729 & &= 727^2 p \mod 729 \\
727(727p-1) &\mod 729 = 727^2 p - 727 \mod 729 & &= 727^2 p + 2 \mod 729 \\
727(727p) - 1 &\mod 729 = 727^2 p - 1 \mod 729 & &= 727^2 p - 1 \mod 729 \\
727(727p-1) - 1 &\mod 729 = 727^2 p - 727 - 1 \mod 729 & &= 727^2 p + 1 \mod 729
\end{aligned}$$

This occupies the region $727^2 p + k$ for $k \in [-1, 2]$. At the next intermediate state, we extend each of these differentials to find the beginning of the new changed region. Given the region $727^2 p + k$ for a range $k \in [a, b]$, the new region is of the form $727^3 p + j$ for $j \in [-2b-1, -2a]$. This thus yields the recurrence:

$$\begin{aligned}
a_0 &= 0 \\
b_0 &= 0 \\
a_i &= -2b_{i-1} - 1 \\
b_i &= -2a_{i-1}
\end{aligned}$$

Note that because $727 \mod 729 = -2 \mod 729$ and the differential may expand to two adjacent positions ("filling in the gaps" between the entries in the scaled version original region), this resulting region is contiguous and has size $\min(2^\ell, 729)$ by induction. The process of solving this recurrence is tedious and non-instructive, so we do not reproduce the math here. However, we note that the final starting position after ℓ states is:

$$p'_\ell = \frac{((-1)^\ell (1+6p) - 3) \cdot 2^{\ell-1} + 1}{3} \mod 729$$

This will allow us to analytically solve for position constraints in later attacks.

We further note that this function is bijective for fixed ℓ and $p : [0, 729)$. Our expression can be written $c_\ell - 2^\ell p \mod 729$ for some ℓ-dependent constant c_ℓ. Thus, so long as $2^\ell \mod 729$ has an inverse modulo 729, the function is bijective. We have $\gcd(2^\ell \mod 729, 729) = \gcd(2^\ell, 3^6) = 1$ and thus the inverse always exists. ■

4 Attacks

We present several attacks on `Curl` below. Note that because IOTA used `Curl` to hash transactions of fixed size, 2673 trytes [11], length extension attacks as in Sections 4.1 and 4.2 could not be used to launch a realistic attack on the live IOTA network. Nevertheless, these are undesirable hashing behaviors and may indicate further vulnerabilities. However, the attacks in Sections 4.4 and particularly 4.5 are relevant for messages of fixed length and form a plausible attack vector.

For clarity, we define $\texttt{Curl}_n(x; s)$ to be the length-n hash of x provided the initial state s. n and s are optional parameters, with default values 243 and 0^{729} if unspecified. Let $S(x)$ be the sponge state s after absorbing the message x (without squeezing to generate any output). Note that with this notation, we can decompose the hashing process into chunks with multiples of $|r|$ size (plus a final block of size $\leq |r|$), as in $\texttt{Curl}_n(x \| y) = \texttt{Curl}_n(y; S(x))$. Finally, by $x[i:j]$ we mean the substring of x from index i (zero-indexed) up to and excluding index j.

4.1 Padding Attack

We formalize an attack proposed in [7], where we can create second-preimage collisions for inputs of length not an integer multiple of $|r|$.

Attack 1 (Identity-Mapped Padding)**.** *For any prefix p of length integer multiple of $|r|$, if $|m| > 1$, then for all integers $k \leq 243$, $\texttt{Curl}_n(m; S(p)) = \texttt{Curl}_n(m \| S(p)[|m| : k]; S(p))$.*

Proof. By the definition of `transform`, the output is uniquely determined by the input state $s = r\|c$. There are two cases:

> For $\texttt{Curl}_n(m; S(p))$, this input state is $m\|S(p)[|m|:729]$ by the update rule in Section 3, because m is superimposed on top of the existing state.
>
> For $\texttt{Curl}_n(m\|p[|m|:k]; S(p))$, this input state is $m\|S(p)[|m|:k]\|S(p)[k:729]$.

Because the input states to `transform` are identical, we can use length extension to create collisions in the last block of any message with length that is not an integer multiple of $|r| = 243$.

■

4.2 Zero-Extension Full-State Collision

We now describe a straightforward full-state collision in `Curl`. Such a state collision in `Curl` was first noted in [7]. A full-state collision is a set of plaintexts P such that for all $p, p' \in P$, $S(p) = S(p')$. Note that $S(p) = S(p')$ implies that $\texttt{Curl}_n(p) = \texttt{Curl}_n(p')$ for all n. We further require that all plaintexts $p \in P$ are of length $243k$ for some integer k. This is easily achievable using the approach described in Section 4.1. This requirement ensures that the state of the sponge after absorbing the full-state collision plaintext is not affected by any values that may be appended. This condition is necessary for the proof of Theorem 2.

The `Curl` hash function possesses the interesting design feature that $\texttt{Curl}_n(0^k) = 0^n$ for all $k \geq 0$. Thus hashing the zero-string always yields the zero-string.[3] This implies that $\texttt{Curl}_n(0^{243k}) = \texttt{Curl}_n(0^{243j})$ and $S(0^{243k}) = S(0^{243j})$ for all $k, j \geq 0$. To see why this is the case, we note that at each round, `Curl`'s S-box performs the mappings $(0,0) \to -1$, $(-1,-1) \to 1$, and $(1,1) \to 0$. As the sponge is initialized to the string 0^{729} and it remains unchanged after the first 243 values are substituted with our input, after a single round, the sponge state will be $(-1)^{729}$. After two, the sponge will be entirely 1. After three, the sponge is again entirely 0. Thus, because the round count is a multiple of 3, our sponge arrives at the state 0^{729} unchanged. By induction, this continues as we absorb additional sets of 243 zero-trits. Thus, this property immediately admits a full-state collision: the set of plaintexts $P_0 = \{0^{243k}\}_{k=0}^{\infty}$.

More significantly, given this full-state collision, we can generate an unbounded number of derivative collisions.

Theorem 2 (Extension of Full-State Collisions)**.** *Given a full state collision P and a message m, for all $p, p' \in P$, $\texttt{Curl}_n(p\|m) = \texttt{Curl}_n(p'\|m)$.*

Proof. The proof of this theorem is trivial given the definition of the absorb stage. We have, per the definition of a full-state collision that $S(p) = S(p')$ for all $p, p' \in P$. As $|p| = 243k$ for some integer k, p will be fully absorbed after k iterations independently of m. The same analysis holds for p'. Thus $\texttt{Curl}_n(p\|m) = \texttt{Curl}_n(m; S(p))$ and $\texttt{Curl}_n(p' \circ m) = \texttt{Curl}_n(m; S(p'))$. However, as $S(p) = S(p')$, these values are identical and the messages collide. ■

Given this property, we can stage a second-preimage attack on `Curl`, allowing us to generate a message m' such that $\texttt{Curl}_n(m') = \texttt{Curl}_n(m)$ for a given m.

Attack 2 (Second-Preimage Attack Using Zero-Extension)**.** *Given a plaintext m and hashed value $\texttt{Curl}_n(m)$, for all $p \in P_0$, $\texttt{Curl}_n(p\|m) = \texttt{Curl}_n(m)$.*

Proof. As P_0 is a valid full-state collision, per Theorem 2 we have that $\texttt{Curl}_n(p\|x) = \texttt{Curl}_n(p'\|x)$ for all $p, p' \in P_0$. However, as the empty string $\varnothing \in P_0$, we thus have $\texttt{Curl}_n(p\|x) = \texttt{Curl}_n(\varnothing\|x) = \texttt{Curl}_n(x)$. Thus $p\|x$ is the desired second input. ■

[3] IOTA purports that this feature makes it easier to recognize when zero or empty strings have been hashed, but we note that because the hash function is deterministic, the zero-string would always have an identifiable hash value [9]. Furthermore, it is not abundantly clear that this was a meaningful design choice for the hash function. Note that for all choices of S-box, $\texttt{Curl}_{243}(0^{243}) = y^{243}$ for some trit y. We can force $y \neq 0$ through some slight modifications to the S-box, but this necessarily admits another type of length extension attack where our full-state collision is the set $\{0^{243} \circ y_0^{243} \circ y_1^{243} \circ \cdots \circ y_k^{243}\}_{k=0}^{\infty}$. See 5.2 for a full consideration of these issues.

4.3 A Related Digest Attack

Due to the relatively simple cyclic structure used within `transform`, a related digest can be generated by trit-shifting certain inputs. This attack was first discovered by Heilman et al. [7].

Attack 3 (Related Digest Generation through Trit-Shift Operation)**.** *For all messages m and a, b, k such that $a + |m| + b = 243$, $\texttt{Curl}_n(0^a \| m \| 0^b) \approx \texttt{Curl}_n(0^{a-k} \| m \| 0^{b+k}) \gg l \approx \texttt{Curl}_n(0^{a+k} \| m \| 0^{b-k}) \ll l$, for some l, where $\gg$ and $\ll$ denote trit shift and $\approx$ denotes equality, except at the first and last k positions.*

Proof. We first note that because modular addition is used to iterate over the positions in the previous sponge state, adjacent positions in this cyclic list of positions are used to compute S-boxes. When we trit shift a message that is padded by zeroes on both sides, we trit shift the entirety of the initial sponge state, as the remainder is initialized to zero.

The trit at position p is used in the computation of the new trits at positions $727p \mod 729$ and $727p - 1 \mod 729$. After a shift of one bit, these become $727(p-1) \mod 729 = 729p + 2 \mod 729$ and $727(p-1) - 1 \mod 729 = 729p + 1 \mod 729$. Critically, order is retained, but an offset is applied. This corresponds to a circular shift at each step of `transform`. Thus, the final digest is a trit-shift of the original digest, with unpredictable values outside the range of this shift. ■

4.4 A Constructive Collision Attack

We now formalize the differential cryptanalysis approach to constructing a collision set forward by Heilman et al. [6].

Attack 4 (Constructive Collision Generation through Differential Cryptanalysis)**.** *Through a computationally feasible constraint problem and a brute force search over 12 rounds of $\texttt{Curl}$, a novel collision can be constructed in minutes on commodity hardware.*

Proof. We first select a trit position we would like to toggle between two messages. Our algorithm will then produce two colliding messages that are identical except for the toggled trit. The algorithm proceeds in two phases: a constraint phase and a brute force phase.

During the constraint phase, we exploit two undesirable properties of the `transform` function: slow propagation of differentials and simple algebraic structure. We utilize Lemma 2, which shows that at any state where only one differential is present, the next state can have at most two differentials. Thus, at each round of `transform`, we can at most double the number of differentials in our state. As we have introduced a single differential into our initial state by toggling a trit in our message, we must limit the spread of this differential.

Fortunately, the `Curl` S-box has useful properties for avoiding the propagation of this differential. Due to the modular structure used for computing `permute`, the differential will appear in the first position of one `permute` operation and the second position of another. If the differential appears in the first position, we cannot stop a new differential from occurring, as no second argument allows the output to remain unchanged if the first input changes. However, if the differential appears in the second position, a first input of -1 allows the differential $\{-1, 0\}$, and a first input of 1 allows the differential $\{0, 1\}$.

			y	
		-1	0	1
	-1	1	1	-1
x	0	0	-1	1
	1	-1	0	0

Figure 7: Propagation of differentials in `permute`.

Thus, if we wish to limit the propagation of a differential throughout the rounds of `transform`, we simply express each round as a recursive function of S-boxes and ensure that when a differential occurs in the second input of an S-box, the first input is constrained such that the differential does not propagate. This creates a set of constraints that guarantee that any message that satisfies the constraints will not propagate the differential. Note that these constraints will constrain both positions in the message and positions in the initial state of the sponge. As such, we must brute force an input message that generates such a state.

Unfortunately, due to the exponential blowup of these constraints, this is only feasible for a small number of rounds. In our experimentation, 8 could be achieved on the order of minutes, while larger numbers of rounds took on the order of hours. However, once a match is found, we have a message that can be modified at a large set of positions such that the differential at our fixed position will not propagate for 8 rounds.

At this point, it is instructive to consider what constitutes a collision for the hash function. Because the digest is simply extracted from the first 243 trits of the sponge state, we must guarantee that after all 27 rounds, all differentials lie in the latter two-thirds of the sponge state, which is of size 486. To do so, we invoke Lemma 3, which guarantees that ℓ rounds after a state with only one differential, the number of differentials is at most 2^{ℓ}. We note that after 7 rounds, at most $2^7 = 128$ differentials exist, all in a contiguous region. So long as this region lies entirely within the latter two-thirds of the sponge state, we have produced a collision. Thus, we aim to find a message satisfying the precomputed constraints that contains only one collision after an additional 12 rounds of hashing. If this is the case, with high probability the entire differential region will lie in the sponge state not used to produce the digest, creating a collision. Note that there are a total of $486 - 128 + 1 = 359$ offsets at which the final differential may begin so as to lie completely within the latter two-thirds of the sponge state. As the position of the differential after ℓ states as computed in Lemma 3 is bijective in p, the original differential position, our final differential will lie in the desired range with probability $231/729 \approx 0.492$.

This brute force search is feasible on commodity hardware due to the relatively low probability of differential propagation at each round. In our experimentation, this took several minutes on commodity hardware. ∎

Using this methodology, we were able to find the following non-trivial collisions, among others. Note that the digests are identical despite the messages differing at a single trit.

```
ACMUXEIFDOIVQMVZNXPNWGSA9JGCN9RIMWOYNFLAVLBKRJPKRAYFCGSD9CAJEFVPHIWRZEKQHUHCAKKSTXM
DZMMVEVVCTQFRTMDR9QLPG9QUWBHBQBVOPDWDIOFUWBK9IREKOUVRHDODLLXCLMJWZZXENYXDUSVDGU
hash = BUEXRNXFUP9HUMBOJWJZBQKDTZKOUVUXSJAXGKMNH9I9EWNBXPBCFNEPBFCQFDYZZCBMXOTP9DOIMKEZ9

ACMUXEIFDOIVQMVZNXPNWGSA9JGCN9RIMWOYNFLAVLBKRJPKRAYFCGSD9CAJEFVPHIWRZEKQHUHCAKKSTXM
DZMMVEVVCTQFRTMDR9QLPG9QUWBHBQBVOPDWDIOGUWBK9IREKOUVRHDODLLXCLMJWZZXENYXDUSVDGU
hash = BUEXRNXFUP9HUMBOJWJZBQKDTZKOUVUXSJAXGKMNH9I9EWNBXPBCFNEPBFCQFDYZZCBMXOTP9DOIMKEZ9
```

As part of this work, we have created an open-source implementation of the tools used to construct a novel collision in `Curl`. To the best of our knowledge, no such tools were publicly available before this report. Additional collisions and an open-source implementation of the colliding algorithm are available at curl-collisions.

4.5 A Constructive Full-State Collision

We next build on Attack 4 to create a full-state collision.

Attack 5 (Constructive Full-State Collision through Differential Cryptanalysis)**.** *Through slight modifications to Attack 4, we can produce a full-state collision in the sponge state, enabling generation of an arbitrary number of collisions.*

Proof. Fortunately, the extension from the singular collisions generated in Attack 4 to full state collisions is rather trivial. Consider the final step of the algorithm, in which we search for a message pair that generates differentials only in the latter two thirds of the sponge state. If, instead, these differentials occurred only in the first third of the sponge state, the messages would not collide.

However, if we append 243 trits to the ends of our messages, this region will be entirely overwritten, discarding the differentials and creating a full-state collision. Thus, we amend our search from above to seek differentials in only the first 243 trits of the sponge state. We then append an arbitrary string of 243 trits to both messages and create a full-state collision. Given the conditions described in Attack 4, this occurs when the differential lies at $243 - 127 + 1 = 117$ locations, which occurs with probability $117/729 \approx 0.160$. ∎

Using this methodology, we were able to construct the following full state collision prefix, excluding the arbitrary 243 trits on the end. This search took on the order of an hour on commodity hardware.

```
PJGZBOAWTZGMTXBKRFQJMBWNPIKNPMBNHFMMPDZGVB9XXJ9MCJAIQKXAHRMGCWN9XL9SIYZC9TUGFEBSK
9GBNRYYAXXRWCPGTZR9XRIHXMYGRTEAHUSYGVKDSAUW9VTVJMMJXRLZRNZRPMEOFYAVBTHM9GZYEGOWQT

PJGZBOAWTZGMTXBKRFQJMBWNPIKNPMBNHFMMPDZGVB9XXJ9MCJAIQKXAHRMGCWN9XL9SIYZC9TUGFEBSK
9GBNRYYAXXRECPGTZR9XRIHXMYGRTEAHUSYGVKDSAUW9VTVJMMJXRLZRNZRPMEOFYAVBTHM9GZYEGOWQT
```

Full-state collisions are automatically detected and reported in our open-source implementation.

5 Structural Analysis

We now consider the attacks described above in the general model used by `Curl`: a sponge construction in which input overwrites sponge state and a permutation-substitution network is used to absorb input. Let the initial state of the sponge be given by $s_0 : \{-1,0,1\}^{729}$. Let the S-box used be given by $f : \{-1,0,1\}^2 \to \{-1,0,1\}$. These analyses demonstrate that our attacks show weaknesses in the structure of `Curl` itself, not just the constants and parameters selected by IOTA.

5.1 Full Parameter Invariance of Attack 1

We now consider a variant of `Curl` in which all semantics are identical except for a substituted S-box, f. We note that our proof for Attack 1 remains unchanged in this setting, as it has no dependency on the S-box. Intuitively, if we pad our message with the corresponding values in the sponge state, the state is unchanged after the message chunk is copied in. This is true regardless of the S-box used.

Furthermore, this attack succeeds for any initialization of the sponge s_0. The sponge state after hashing a message is easily computable and deterministic regardless of the initial sponge state s_0. Thus, we can always pad with trits that do not result in a changed sponge state.

5.2 S-Box Invariance of Attack 2

Consider again the generalized version of `Curl` with S-box f. Our zero-extension full-state also holds in this setting with minor modifications. Suppose our S-box specifies the mappings $f(-1,-1) = a_{-1}$, $f(0,0) = a_0$, and $f(1,1) = a_1$. Our sponge begins in the state 0^{729}, proceeds to the state a_0^{729}, then to the state $a_{a_0}^{729}$, recursively for 27 states. We can represent this as a state machine in which we begin in state 0, then proceed to state a_0, then proceed to state a_{a_0}, and so on. The state specifies the value of every entry in the sponge state.

If the transitions in this state machine form a cycle of length 3, after 27 transitions we will be at our original state 0, and the attack is unchanged. Otherwise, we may arrive at a state other than 0. In this case, we simply extend our message with 243 trits of the value we arrived at. As our state machine must contain a cycle (as it has three nodes and three edges), we will end up at a new state. If this state is 0, we have produced a full state collision for $\{0^{243}a_0^{243}, 0^{243}a_0^{243}0^{243}a_0^{243}, \dots\}$. If the state is not zero, we simply follow the cycle until it becomes periodic (with period 1 or 2), and produce a set of colliding messages that adheres to that pattern. This attack is a special case of 1.

Note that while we can always produce a full-state collision regardless of f, the second-preimage attack only succeeds when the state machine has a cycle of length 3. Otherwise, our message leaves the sponge in a different state than it was upon initialization and we cannot use it to extend arbitrary messages. However, if the S-box does not admit a cycle of length 3 then `transform` is necessarily not a permutation.

5.3 S-Box Invariance of Attack 3

Attack 3 has no dependency on the S-box and relies only on the cyclic structure of the `transform` function and the state s_0. As such, so long as the state s_0 is initialized to k^{729} for some k, we can produce a similar related digest attack by padding our message with k instead of 0, regardless of the choice of S-box.

5.4 Initialization Invariance of Attacks 4 and 5

Next, consider a generalized version of `Curl` in which the initial sponge state is s_0 but the S-box remains identical. We note that this does not pose a problem for our constructive collision attacks, as we determine a message that satisfies our propagation constraints by brute force. In the new variant, we simply perform this search with an initial state s_0 instead of 0^{729}. After finding a state that satisfies the constraints, the remainder of the attack remains identical.

In the case that the S-box is not identical, we can use the same form of the attack with different constraints, assuming the S-box still has duplicates in its rows or columns. If the S-box does not contain duplicates in its rows or columns (and thus $f(\cdot, x)$ and $f(x, \cdot)$ are bijective for all x), the attack no longer succeeds, as we cannot easily constrain the propagation of differentials.

6 Signatures

While the details of IOTA's signature scheme remain murky, we can see from the reference implementation that signatures act on the hashes of transaction bundles, rather than on the bundles themselves [8]. This, combined with the failure of `Curl`'s collision resistance property, opens IOTA up to an EU-CMA (Existential Unforgeability Chosen Message Attack) [5].

At a high level, this attack game requires the adversary to produce a valid signature for any m given a polynomial number of signing queries for messages $m' \neq m$. Because IOTA's scheme signs $h(m)$ rather than m itself, a signature valid for a queried m' will be identical to a signature for m such that $h(m) = h(m')$. This corresponds exactly to the Collision Resistance property of cryptographic hash functions; because we can efficiently find a colliding pair m, m' for `Curl`, in the EU-CMA attack game we can query a signature for m' and output it as a forged signature for the message m. Therefore, contrary to IOTA's claims, `Curl`'s Collision Resistance is required for the EU-CMA security of its signature scheme.

Though existential forgery and chosen message attacks may seem like implausible attack models, see the original [7] for discussion of how they might be applied for practical attacks on the IOTA network.

7 Conclusion

In this paper, we have demonstrated, formalized, and made publicly available practical second-preimage attacks on `Curl` by length extension and fixed-length/unbounded-length collision resistance attacks by differential cryptanalysis. Our analysis suggests that these vulnerabilities are not due to the concrete parameters used to instantiate `Curl` in the implementation of IOTA, like `permute` or $|r|$, but rather fundamental problems with the new variants of the sponge functions `absorb` and `transform`.

8 Acknowledgements

We would like to thank Boaz Barak for his feedback, support, and assistance in simplifying the proof of Theorem 1.

References

[1] Bertoni, Guido, et al. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications." *Selected Areas in Cryptography Lecture Notes in Computer Science*, 2012, pp. 320–337.

[2] Biham, Eli, and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, 2011.

[3] Cochran, Martin. "Notes on the Wang Et Al. 2^{63} SHA-61 Differential Path." pp. 1–19. Cryptology EPrint Archive, 2007, https://eprint.iacr.org/2007/474.

[4] "CoinMarketCap", 6 Aug. 2017, web.archive.org/web/20170806014139/https://coinmarketcap.com/.

[5] Goldwasser, Shafi, et al. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks." *SIAM Journal on Computing*, vol. 17, no. 2, 1988, pp. 281–308.

[6] Heilman, Ethan. "Breaking IOTA's Curl Hash Function." 3 May 2018, www.youtube.com/watch?v=x3W0TYbLk4U.

[7] Heilman, Ethan, et al. *IOTA Vulnerability Report: Cryptanalysis of the Curl Hash Function Enabling Practical Signature Forgery Attacks on the IOTA Cryptocurrency.* 7 Sept. 2017, https://github.com/mit-dci/tangled-curl/blob/master/vuln-iota.md.

[8] "Iotaledger/Iota.lib.java." *GitHub*, IOTA, https://github.com/iotaledger/iota.lib.java/blob/f43d606f041d1bf6eceb44d6758b710149b15d0a/src/main/java/jota/utils/Signing.java#L186.

[9] Ivancheglo, Sergey, et al. Leaked Email Chain between IOTA Foundation members and MIT Digital Currency Initative researchers. July–October 2017, http://www.tangleblog.com/wp-content/uploads/2018/02/letters.pdf.

[10] Rogaway, Phillip, and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance." *Fast Software Encryption Lecture Notes in Computer Science*, 2004, pp. 371–388.

[11] Schiener, Dominik. "The Anatomy of a Transaction." *IOTA Guide*, domschiener.gitbooks.io/iota-guide/content/chapter1/transactions-and-bundles.html.

[12] *Security Alert.* Parity Technologies Ltd., 8 Nov. 2017, paritytech.io/security-alert-2/.

[13] Sønstebø, David. "Upgrades & Updates." *IOTA Blog*, The IOTA Foundation, 7 Aug. 2017, blog.iota.org/upgrades-updates-d12145e381eb.

[14] Wang, Xiaoyun, and Hongbo Yu. "How to Break MD5 and Other Hash Functions." *Lecture Notes in Computer Science Advances in Cryptology – EUROCRYPT 2005*, 2005, pp. 19–35.