Analysis of the Kupyna-256 Hash Function

Christoph Dobraunig, Maria Eichlseder, and Florian Mendel

Graz University of Technology, Austria
christoph.dobraunig@iaik.tugraz.at

Abstract. The hash function Kupyna was recently published as the Ukrainian standard DSTU 7564:2014. It is structurally very similar to the SHA-3 finalist Grøstl, but differs in details of the round transformations. Most notably, some of the round constants are added with a modular addition, rather than bitwise xor. This change prevents a straightforward application of some recent attacks, in particular of the rebound attacks on the compression function of similar AES-like hash constructions. However, we show that it is actually possible to mount rebound attacks, despite the presence of modular constant additions. More specifically, we describe collision attacks on the compression function for 6 (out of 10) rounds of Kupyna-256 with an attack complexity of 2^{70}, and for 7 rounds with complexity $2^{125.8}$. In addition, we can use the rebound attack for creating collisions for the round-reduced hash function itself. This is possible for 4 rounds of Kupyna-256 with complexity 2^{67} and for 5 rounds with complexity 2^{120}.

Keywords: hash functions · cryptanalysis · collisions · free-start collisions · Kupyna · rebound attack

1 Introduction

Recently, Oliynykov et al. [11] published an English specification of the new Ukrainian hash standard DSTU 7564:2014, also known as Kupyna. In contrast to the previous standard GOST 34.311-95, the new hash standard facilitates more effective software implementations. Of course, it is also intended to offer improved security compared to the old GOST standard, which has shown weaknesses against collision attacks [7]. As Kupyna is a national standard, it is likely to find wide-spread adoption in the Ukraine. Thus, comprehensive third-party analysis is necessary to evaluate the resistance of Kupyna against cryptanalytic attacks.

The Kupyna design aims to achieve a high level of security by relying on well-known and well-analyzed building blocks. It shares a notable similarity with the SHA-3 finalist Grøstl [2]. Kupyna's mode of operation, in particular its compression function, is nearly identical to the one used in Grøstl, and its permutations – though different – follow very similar design ideas. One of Kupyna's two permutations employs the wide-trail design strategy [1] of AES. Therefore, this permutation shares a common basis with Grøstl's permutations, although Kupyna uses other constants, S-boxes, rotation values, and a different MDS

matrix. The second Kupyna permutation differs from the first in the constant addition, which applies addition modulo 2^{64} rather than bitwise xor. This modular addition serves to differentiate the two permutations, but can also be seen as a measure to complicate algebraic cryptanalysis. Furthermore, it implies additional relations over byte boundaries. As a consequence, the modular addition leads to a weaker alignment for differential trails, making statements about the minimum number of active S-boxes in a differential trail more complicated, since the linear layer no longer achieves an optimal branch number.

Our contribution. In this paper, we provide the first third-party analysis of the new Ukrainian hash standard Kupyna. We present collision attacks on round-reduced variants of Kupyna-256 for up to 5 out of 10 rounds, and collisions for the compression function of Kupyna-256 for up to 7 rounds. A summary of our results can be found in Table 1.

Table 1. Overview of collision attacks on Kupyna-256.

Hash function	Target	Rounds	Complexity	Reference
Kupyna-256	Compression function	6	2^{70}	Sect. 3
		7	$2^{125.8}$	
	Hash function	4	2^{67}	Sect. 4
		5	2^{120}	

Our attacks make use of the capability of rebound attacks [8] to efficiently generate pairs of values which follow a given truncated differential trail. The core idea of such a rebound attack is to create many solutions with a low complexity per solution during the inbound phase, and propagate those solutions in a probabilistic manner during the so-called outbound phase.

To create solutions with a low average complexity during the inbound phase, the rebound attack takes advantage of the strong alignment of truncated differential trails and the underlying independence of parts of the cipher. As mentioned before, one of the two permutations of Kupyna does not provide such a strong alignment. Hence, it is not trivial to perform the rebound attack for this permutation. However, in this work, we show how to deal with modular constant additions during the inbound and the outbound phase to be able to perform rebound attacks on such constructions.

Related work. Due to the high similarity of Kupyna with Grøstl, the fundament of our attacks is the analysis of Grøstl. For Grøstl, the best attacks are based on the rebound attack and its improvements [3, 5, 6]. Distinguishers for round-reduced variants of the Grøstl permutation were published in [4]. Rebound attacks leading to collisions for the highest number of rounds for the compres-

sion function of Grøstl were shown in [9]. An efficient collision attack covering 5 rounds of the hash function Grøstl itself was published in [10].

In the meantime, Zou and Dong [13] independently analyzed the Kupyna hash function, and observed the applicability of some of the attacks on Grøstl to Kupyna. They present pseudo-preimage attacks on 6 rounds of Kupyna-256 and 8 rounds of Kupyna-512 with time complexities $2^{250.33}$ and $2^{498.33}$, respectively, which are essentially identical to the original Grøstl attacks [12]. Additionally, they also noted the hash function attack on 5 rounds very similar to Section 4 of this paper.

Outline. The remainder of the paper is organized as follows. First, we start with a short description of Kupyna in Section 2. Next, we show how to apply the rebound attack on round-reduced versions of the compression function of Kupyna-256, to create semi-free-start collisions for 6 and 7 rounds, in Section 3. Then, we apply a collision attack for 4 and 5 rounds of Kupyna-256 in Section 4. Finally, we conclude in Section 5.

2 Description of Kupyna

Kupyna [11] is a family of iterated hash functions defined in the Ukrainian standard DSTU 7564:2014. The design principles of Kupyna are very similar to the SHA-3 finalist Grøstl [2]. As in Grøstl, the compression function of Kupyna is built from two distinct permutations $T^{\oplus}$ and T^{+}, which are both based (to a certain degree) on the AES design principles. In the following, we describe the components of the hash function in more detail.

2.1 The Hash Function

The Ukrainian standard DSTU 7564:2014 defines two main variants, Kupyna-256 and Kupyna-512, which produce a hash output size of $n = 256$ and $n = 512$ bits, respectively (the third recommendation, Kupyna-384, is simply a truncated version of Kupyna-512). The hash function first pads the input message M and splits the message into blocks $m_1, m_2, \ldots, m_t$ of ℓ bits each, with $\ell = 512$ for Kupyna-256 and $\ell = 1024$ for Kupyna-512. The message blocks are processed via the compression function $f(h_{i-1}, m_i)$, which updates the internal ℓ-bit chaining value h_i, and an output transformation $\Omega(h_t)$ to produce the final hash value h:

$$\begin{aligned} h_0 &= \mathrm{IV} \\ h_i &= f(h_{i-1}, m_i) \quad \text{for } 1 \le i \le t \\ h &= \Omega(h_t). \end{aligned}$$

The compression function f is based on two ℓ-bit permutations $T^{\oplus}$ and T^{+} and is defined as follows (see also Fig. 1):

$$f(h_{i-1}, m_i) = T^{\oplus}(h_{i-1} \oplus m_i) \oplus T^{+}(m_i) \oplus h_{i-1}.$$

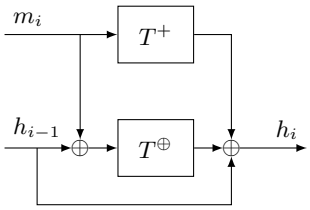

Fig. 1. The compression function $h_i = f(h_{i-1}, m_i)$ of the Kupyna hash function, using ℓ-bit permutations $T^\oplus$ and T^+.

The output transformation Ω is applied to h_t to give the final hash value h of size n, where $\text{trunc}_n(x)$ discards all but the most significant n bits of x:

$$\Omega(h_t) = \text{trunc}_n(T^\oplus(h_t) \oplus h_t).$$

2.2 The Permutations $T^\oplus$ and T^+

In the remaining document, we focus our analysis on Kupyna-256. The structure of the two permutations $T^\oplus$ and T^+ of Kupyna is very similar to the ones of Grøstl. As in Grøstl-256, each Kupyna-256 permutation updates an 8×8 state of 64 bytes in 10 rounds. In each round, the round transformation updates the state by means of the sequence of transformations

$$\mathsf{MixBytes} \circ \mathsf{RotateBytes} \circ \mathsf{SubBytes} \circ \mathsf{AddConstant}.$$

In the following, we briefly describe the round transformations of $T^\oplus$ and T^+ in more detail. Note that the Kupyna specification [11] refers to the transformations as ψ, $\tau^{(\ell)}$, π', and $\kappa_i^{(\ell)}/\eta_i^{(\ell)}$, respectively, but we use the more commonly understood AES-like transformation names in the remaining document.

AddConstant (AC). In this transformation, the state is modified by combining it with a round constant. This is the only transformation where the two permutations differ. While $T^\oplus$ combines the round constant with each column with bitwise xor ($\oplus$), T^+ applies column-wise modular addition mod 2^{64} ($+$). The round constants for $T^\oplus$ are defined as follows for round r, $1 \leq r \leq 10$, and column j, $0 \leq j < 8$:

$$\omega_j^{(r)} = ((j \ll 4) \oplus r, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00}, \mathtt{00})^\top.$$

The (round-independent) round constants for T^+ for column j are given by:

$$\varsigma_j^{(r)} = (\mathtt{F3}, \mathtt{F0}, \mathtt{F0}, \mathtt{F0}, \mathtt{F0}, \mathtt{F0}, \mathtt{F0}, (7-j) \ll 4)^\top,$$

where the first byte `F3` and the first row of the state serve as the least significant bytes for the addition. This modular addition performed in T^+ is also the main

difference of Kupyna compared to Grøstl from a cryptanalytic point of view. Note that this modular addition destroys the columnwise optimal branch number of the linear layer. We give an example with only 6 (rather than 9) active S-boxes over 2 rounds in Appendix A.

SubBytes (SB). The SubBytes transformation is the same for $T^{\oplus}$ and T^{+}. It is a permutation consisting of S-boxes applied to each byte of the state (with 4 different S-boxes, depending on the row index). The 8-bit S-boxes are designed to provide good cryptographic properties against differential and linear attacks. For a detailed description of the S-boxes, we refer to the specification [11]. Note that the SubBytes transformation is the only non-linear transformation of the permutation $T^{\oplus}$.

RotateBytes (RB). The RotateBytes transformation is a byte transposition that cyclically shifts the bytes of each state row by different offsets: row j is shifted rightwards by j byte positions, $0 \leq j < 8$. This transformation is the same for both permutations $T^{\oplus}$ and T^{+}. This is also in contrast to Grøstl, where two different sets of rotation constants are defined for the two permutations, in order to diversify between the two. In Kupyna, this role is solely played by AddConstant.

MixBytes (MB). The MixBytes transformation is a permutation operating on the state column by column. To be more precise, it is a left-multiplication by an 8×8 circulant MDS matrix over $\mathbb{F}_{2^8}$. The coefficients of the matrix are determined in such a way that the branch number of MixBytes (the smallest nonzero sum of active input and output bytes of each column) is 9, which is the maximum possible for a transformation with these dimensions. This transformation is the same for both permutations $T^{\oplus}$ and T^{+}.

3 Semi-free-start Collisions for 6 and 7 Rounds

In this section, we mount a collision attack on 6 rounds of the compression function of the Kupyna-256 hash function. The attack described here is based on the rebound attacks on Grøstl [9] using SuperBox matches [3,5,6]. Hence, the high-level attack strategy to create pairs following differential truncated trails stays the same. Due to the round-constant addition modulo 2^{64} in the permutation T^{+}, a straightforward application of the Grøstl attack is not possible, and some additional considerations are required.

Finally, in Section 3.5, we also show how to extend the collision attack to 7 rounds of the compression function, also based on rebound attacks on Grøstl [9].

3.1 Attack Strategy

We target differential trails similar to those in [9]. The core idea of this attack is to use the same truncated differential trail in both permutations $T^{\oplus}$ and T^{+}.

If the differences at the input and the output match, we get a semi-free-start collision. Note that the differences are introduced by the message block m_i, whereas the chaining value h_{i-1} is free of differences, but can also be chosen arbitrarily. A high-level view of this attack is illustrated in Fig. 2.

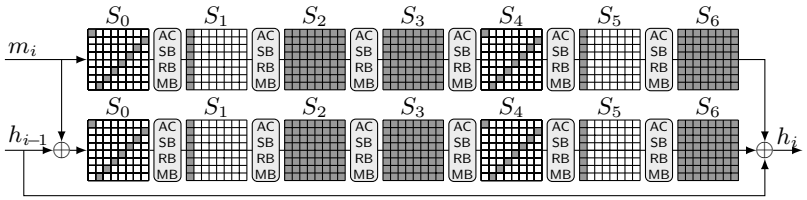

Fig. 2. Collision attack on 6 rounds of the Kupyna-256 compression function.

To find matching trails, we use the rebound attack strategy introduced in [9]. This strategy consists of an inbound and an outbound phase. During the inbound phase, solutions for the core of the trail are deterministically created with a complexity close to 1 per solution, whereas the propagation through the outbound phase is done in a probabilistic manner. This, combined with the fact that we have to match the input and output differences of the independently created trails for $T^{\oplus}$ and T^{+}, suggests a truncated differential target trail which is dense in the middle and gets sparse towards the ends, with the following numbers of active bytes (S-boxes):

$$8 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 8 \xrightarrow{r_6} 64. \tag{1}$$

3.2 Finding Pairs for $T^{\oplus}$

The permutation $T^{\oplus}$ follows the wide-trail design strategy. Even though $T^{\oplus}$ uses different SubBytes, RotateBytes and MixBytes layers than Grøstl, the differential behaviour on byte level is almost identical, so those changes have a very limited influence on the way a rebound attack is applied (e.g., due to the different rotation constants, slightly different truncated trails are used). Hence, the rebound attack on $T^{\oplus}$ can be done very similarly to the Grøstl permutation [9]. Below, we repeat the essential parts of this attack. We use the same notation to denote intermediate states: S_0 is the initial state, S_i denotes the state after round i ($1 \leq i \leq r$), and the intermediate states after AddConstant, SubBytes, RotateBytes, and MixBytes of round i are labelled S_i^{AC}, S_i^{SB}, S_i^{RB}, and $S_i^{\mathsf{MB}} = S_i$, respectively.

The Inbound Phase. For the inbound phase, we use the SuperBox based technique described by Mendel et al. [9]. This phase covers the round transformations of 2.5 rounds, beginning with MixBytes of round 2 (input state S_2^{RB}), and ending with MixBytes of round 4 (output state S_4). The detailed truncated

differential trail for the inbound phase is shown in Fig. 3. The attack works as follows:

1. First, we start backwards from state S_4, which has 8 active bytes. We enumerate all $2^{8 \cdot 8}$ possible bitwise difference patterns at S_4, and deterministically propagate them backwards through the linear $\mathsf{MixBytes} \circ \mathsf{RotateBytes}$ to S_4^{SB}. The resulting 2^{64} difference patterns for S_4^{SB} are stored in a table D_1.
2. Next, we choose a random difference pattern for state S_2^{RB} and deterministically propagate forward through $\mathsf{MixBytes}$ to state S_2.
3. Finally, we have to connect the inputs of 8 S-boxes of state S_2 belonging to one SuperBox to its corresponding output in state S_4^{SB}. We can do this for each of the 8 SuperBoxes independently in the following way:
   (a) Enumerate all possible 2^{64} value pairs for the SuperBox at state S_2 and propagate forward to state S_4^{SB} .
   (b) Store the resulting 2^{64} value pairs at state S_4^{SB} in a table D_2.
   (c) To find solutions for this SuperBox, filter D_2 with the possible differences in table D_1.

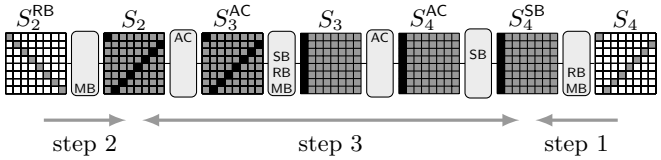

Fig. 3. Inbound phase for $T^{\oplus}$, SuperBox in black

Now we have to determine how many solutions we get for the inbound phase. From a high-level point of view, we have generated $2^{64 \cdot 8}$ pairs of values at state S_2. All those pairs are filtered with the truncated differential of state S_4. Here, 56 bytes need to be zero. Thus, we expect $2^{64 \cdot 8 - 56 \cdot 8} = 2^{64}$ valid solutions (pairs) for the inbound phase. The computational complexity for the inbound phase is 2^{64} round function calls for creating these 2^{64} solutions with a complexity of 2^{64} in memory for storing the tables. The inbound phase can be repeated up to 2^{64} times with other difference patterns at state S_2^{RB}, leading to a maximum of 2^{128} solutions with 2^{128} complexity in time. Hence, the amortized complexity of finding one solution in the inbound phase is 1.

The Outbound Phase. During the outbound phase, the created solutions from the inbound phase have to be probabilistically propagated, first from state S_2^{RB} back to S_0 and then from S_4 to S_6. Both times, they have to follow the pattern of the truncated trail: $8 \xleftarrow{r_1} 8 \xleftarrow{r_{2'}} 8$ ($r_{2'} = \mathsf{RB} \circ \mathsf{SB} \circ \mathsf{AC}$) in the backward direction, and $8 \xrightarrow{r_5} 8 \xrightarrow{r_6} 64$ in the forward direction. Since both trails have a probability very close to one, this phase has only negligible influence on the complexity.

3.3 Finding Pairs for T^+

As for $T^{\oplus}$, we also need to find pairs of values for T^+ which follow the given truncated differential trail. However, the modular constant addition of T^+ on each column of the state can cause difference propagations between bytes, which are impossible in $T^{\oplus}$. Therefore, we cannot rely on the byte alignment of the truncated differential trails anymore. In the following, we show how to handle such uncertainties and how to apply the rebound attack on T^+, although its trails are not strongly aligned. Note that we still target the same truncated trail for $T^{\oplus}$ and T^+ (Fig. 2), and the input/outbound phases cover the same steps (Fig. 3).

The Inbound Phase. The round transformations covered by the inbound phase (states S_2^{RB} to S_4) include two constant additions AC: one between state S_2 and S_3^{AC}, and one between S_3 and S_4^{AC}, both corresponding to step 3 of this phase (see Fig. 3). The other steps 1 and 2 are not influenced by the constant additions and thus, our considerations do not change for them.

The constant addition between S_3 and S_4^{AC} is aligned with each SuperBox. Hence, it only influences each of the 8 SuperBoxes individually. Therefore, the constant addition taking place between state S_3 and S_4^{AC} can be integrated into the SuperBoxes and does not influence our considerations negatively. Unfortunately, this is not the case for the second constant addition.

During step 3, we want to propagate values and differences per SuperBox from its inputs to its outputs. At state S_3^{AC}, the inputs of the SuperBox are represented by bytes in the diagonals of the state. Hence, we cannot integrate the column-wise constant addition that happens before S_3^{AC} into our SuperBoxes anymore. Unfortunately, this constant addition creates a dependence between the 8 SuperBoxes via the carry. To be able to treat the SuperBoxes independently again, we only start with values at state S_2 so that the addition of one of these values with the constant definitely results in a carry for the next byte. By doing so, we can always expect a carry at the input of the bytes for the constant addition (except for the LSB, of course). In this way, we can treat every SuperBox separately and the rest of the attack works as described for $T^{\oplus}$.

Since we have restricted the number of possible values in step 3 to values that generate a carry, we have to determine how many solutions we get. First consider the least significant byte (byte 0), corresponding to constant `F3` and an element x from the first state row. To generate the carry, we require that $x + \texttt{F3} > \texttt{FF}$, so 243 (out of 256) values for x are valid. For the following bytes 1–6, we assume an input carry, so we require that $x + 1 + \texttt{F0} > \texttt{FF}$, which has 241 solutions. Finally, for byte 7, we have no requirements. If we assume that the required bitwise difference for a value pair is uniformly random, then the expected number of valid value pairs for byte 0 is $256 \cdot (1 - 2 \cdot \frac{13}{256} + (\frac{13}{256})^2) \approx 230.6$. Similarly, for bytes 1–6, the expected number of valid pairs is ≈ 226.8. The results are summarized in Table 2.

By using the numbers of Table 2, we see that on average, the number of valid pairs we can create per SuperBox is $(230.6) \cdot (226.8)^6 \cdot 256 \approx 2^{62.8}$. As we

Table 2. Number of byte values and average number of value pairs (for fixed bitwise difference) that produce a carry on modular addition with Kupyna's round constants.

Byte position	Valid values	Valid pairs (average)
Byte 0	243	230.6
Byte 1–6	241	226.8
Byte 7	256	256

have the same filter criterion as for $T^{\oplus}$, we can create $2^{62.8 \cdot 8 - 56 \cdot 8} = 2^{54.4}$ valid solutions for the inbound phase. These solutions are obtained with a complexity of $243 \cdot 241^6 \cdot 256 = 2^{63.4}$, since only this is the number of values per SuperBox that result in carries per byte. We can repeat the inbound phase 2^{64} times to create up to $2^{118.4}$ pairs with a complexity of $2^{127.4}$. In other words, finding one solution in the inbound phase has an amortized complexity of 2^9.

The Outbound Phase. In the outbound phase, the pairs created during the inbound phase are propagated in a probabilistic manner. Since we have to consider the effects of the modular constant addition, the success probability of this phase is reduced compared to $T^{\oplus}$.

First, we want to consider the propagation from state S_2^{RB} back to S_0. Here, we have to follow the truncated trail $8 \xleftarrow{r_1} 8 \xleftarrow{r_{2'}} 8$, where $r_{2'} = \mathsf{RB} \circ \mathsf{SB} \circ \mathsf{AC}$. For the first constant addition between state S_1 and S_2^{AC}, the 8 active bytes lie within one column. Thus, the constant addition will not change the activeness of the bytes with overwhelming probability. The second constant addition occurs between state S_0 and S_1^{AC}, where the active bytes are on the diagonal of the state. Therefore, the constant addition may lead to difference propagation from an active byte to a formerly inactive byte (if one value of the pair produces a carry and the other value does not). Assuming that none of the active bytes receives an input carry (worst case), the probability that none of the 8 active bytes propagates a difference to its neighbouring byte is $(1 - 2 \cdot \frac{13}{256} \cdot \frac{243}{256}) \cdot (1 - 2 \cdot \frac{16}{256} \cdot \frac{240}{256})^6 \cdot 1 \approx 2^{-1.225}$. Summarizing, the probability that a value pair follows the differential trail from S_2^{RB} to S_0 is about $2^{-1.225}$.

For the propagation from state S_4 to S_6, we have another two constant additions, one of which is aligned with the SuperBox. Again, the probability that a pair follows this truncated differential is $2^{-1.225}$. So the probability that a pair created during the inbound phase follows the truncated differentials implied by the outbound phase is $2^{-2.45}$. Combining the inbound and the outbound phase, we can create $2^{54.4-2.45} = 2^{51.95}$ pairs that follow the 6-round truncated differential trail with a complexity of $2^{63.4}$. Hence, one solution that follows the truncated trail for T^{+} can be constructed with an amortized complexity of $2^{11.45}$.

3.4 Results for 6 Rounds

In the last two subsections, we have discussed how to create valid pairs that follow the truncated differential trails for permutations T^{+} and $T^{\oplus}$. To get a

collision for the compression function, we have to find pairs of values for T^+ and $T^\oplus$ such that the input differences and the output differences match. At the input, we have 8 active bytes and therefore 64 bitwise conditions to match. At the output, we can match the state before the linear MixBytes operation, resulting in another 64 bitwise conditions. Due to the birthday paradox, we expect a match on these 128 conditions after creating 2^{64} pairs for T^+ and for $T^\oplus$.

The complexity for creating 2^{64} pairs that follow the truncated trail in $T^\oplus$ is 2^{64}. Creating $2^{64.95}$ pairs which follow the trail in T^+ has a complexity of $2^{76.4}$, settling the total complexity of the attack. A better attack complexity can be achieved by applying an unbalanced birthday attack. Creating 2^{70} pairs for $T^\oplus$ with complexity 2^{70} and $2^{58.55}$ pairs for T^+ with complexity 2^{70} allows us to find a semi-free-start collision for 6 rounds with a total attack complexity of about 2^{70} (time and memory).

Overall, the introduction of the modular addition only increased the attack complexity from 2^{64} to 2^{70} (compared to a variant where T^+ and $T^\oplus$ are essentially identical. Thus, this approach is significantly less effective at preventing this type of rebound attacks than that of Grøstl, namely, using different rotation values in T^+ and $T^\oplus$ to prevent this type of truncated trails from matching at the ends. Note that the low increase of complexity is also due to the special choice of round constants in Kupyna. In particular, our experiments show that if the round constant bytes of $\zeta_j^{(r)}$ were randomly selected, the probability that this kind of attack could succeed with complexity 2^{70} or less is less than 1 in 10 000 (if the constants are still the same for all rounds and columns, as is the case for Kupyna – otherwise, even less).

3.5 Extending the Attack to 7 Rounds

We now extend the previous collision attack from 6 to 7 rounds of the compression function. The attack is based on a closely related truncated differential trail with the following sequence of active S-boxes:

$$8 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 1 \xrightarrow{r_6} 8 \xrightarrow{r_7} 64. \tag{2}$$

Up to round 5 (state S_4 in Fig. 4), this trail is identical to the 6-round attack. Consequently, the whole inbound phase works identically as before, and we only need to adapt the outbound phase in the attack on 7 rounds.

First, we want to determine the probability that a solution created during the inbound phase follows the truncated differential trail for permutation $T^\oplus$. The outbound phase from state S_2^{RB} back to S_0 is the same as for 6 rounds and thus works with a probability of 1. The target trail from state S_4 to S_7 has $8 \xrightarrow{r_5} 1 \xrightarrow{r_6} 8 \xrightarrow{r_7} 64$ active S-boxes. The transitions of $1 \xrightarrow{r_6} 8$ and $8 \xrightarrow{r_7} 64$ active S-boxes have a probability close to 1. The transition of $8 \xrightarrow{r_5} 1$ active S-boxes has a probability of 2^{-56}, which also determines the total probability of the outbound phase for $T^\oplus$.

For the permutation T^+, we have to consider the additional probability that the constant addition does not change the pattern of the truncated differential

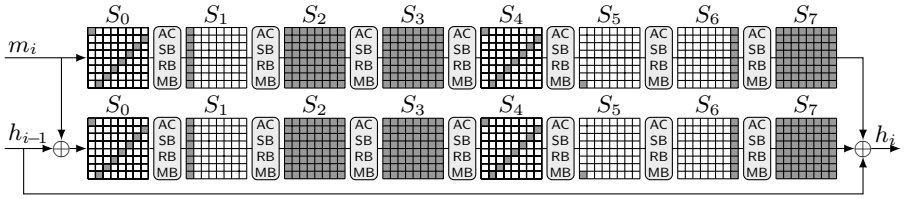

Fig. 4. Collision attack on 7 rounds of the Kupyna-256 compression function.

trail. The probability that the patterns stay the same is $2^{-1.225}$ for the constant addition after S_0 and S_4, and 1 for the constant addition after S_1, S_5 (active byte at the MSB of the constant addition), and S_6. Thus, the probability that the trail during the outbound phase is followed is $2^{-58.45}$ for T^+.

By repeating the attack on $T^\oplus$ 2^s times, $s \leq 64$, we are able to generate up to $2^{s+64-56} = 2^{s+8}$ pairs following the truncated differential trails for $T^\oplus$ with a complexity of 2^{s+64}, and with 2^t repetitions for T^+, up to $2^{t+54.4-58.45} = 2^{t-4.05}$ pairs with a complexity of $2^{t+63.4}$. Additionally, we still have to match an 8-byte condition at the input of the permutations and an 8-byte condition at the output of the permutation. In total, an unbalanced birthday attack with $s \approx 61.8$ and $t \approx 62.4$ gives the best complexity. For $T^\oplus$, we generate $2^{69.8}$ solutions with a complexity of $2^{125.8}$, and for T^+ $2^{58.35}$ solutions with a complexity of $2^{125.8}$. Since $2^{69.8} \cdot 2^{58.35} > 2^{128}$, we expect a match with high probability. This results in a semi-free-start collision over 7 rounds with a total complexity of about $2^{125.8}$ and 2^{70} memory.

4 Collision Attacks on the reduced Hash Function

In this section, we describe collision attacks on Kupyna-256 reduced to 4 and 5 rounds. The attacks are a straight-forward application of the rebound attack on the reduced Grøstl-256 hash function [10] to Kupyna-256. To simplify the description of the attack, we use the alternative description of the hash function, similar to the attacks on round-reduced Grøstl in [10].

4.1 Alternative Description of Kupyna

Let $\hat{T}^\oplus$ and $\hat{T}^+$ denote the permutations $T^\oplus$ and T^+ without the final application of MixBytes. Consider the following alternative description of Kupyna:

$$\begin{aligned}
\hat{h}_0 &= \mathsf{MB}^{-1}(\mathrm{IV}) \\
\hat{h}_i &= \hat{T}^\oplus(\mathsf{MB}(\hat{h}_{i-1}) \oplus m_i) \oplus \hat{T}^+(m_i) \oplus \hat{h}_{i-1} \quad \text{for } 1 \leq i \leq t \\
h &= \Omega(\mathsf{MB}(\hat{h}_t))
\end{aligned}$$

This description of Kupyna is equivalent to the original one by letting h_i be $\mathsf{MB}(\hat{h}_i)$. Just the final MixBytes transformation of the permutations changes

place with the xor operation of the feed-forward. With this modified description, the limited set of differences at the output of the compression function becomes more clearly visible in the attack.

4.2 Attack Strategy

The essential idea of the hash function attack on reduced Kupyna-256 is to have a multi-block attack, but such that all message blocks except the first have no differences. This way, we can concentrate on the trails through $\hat{T}^{\oplus}$, and use the freedom of the message blocks to successively cancel all differences in the internal chaining values.

The first message block can be selected arbitrarily, we only require a difference in the message. This way, we start from some arbitrary difference in the chaining variable for the second block, and want to convert it into an output difference equal to zero after 8 more compression function calls. The corresponding 8 message blocks are fully controlled by the attacker and must not contain any differences. Then, each of the 8 message blocks is used to cancel one eighth of the differences at the output of the compression function to result in a collision at the end (see Fig. 5).

The trails used in our collision attack on 4 and 5 rounds start from a fully active input state and map it to 8 active bytes at the output of $\hat{T}^{\oplus}$:

$$\text{4-round collision:} \quad 64 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8 \xrightarrow{r_3} 8 \xrightarrow{r_4} 8, \tag{3}$$

$$\text{5-round collision:} \quad 64 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8 \xrightarrow{r_3} 1 \xrightarrow{r_4} 8 \xrightarrow{r_5} 8. \tag{4}$$

The trails for $\hat{T}^{\oplus}$ are similar to the trails used in Section 3 and Section 3.5, though this time, just covering the inbound phase and the outbound phase computed in forward direction. Hence, valid pairs can be created using the same methods. For a given input difference, we can construct 2^{64} or 2^8 pairs following trail (3) or trail (4), respectively, with a complexity of 2^{64} using the rebound attack.

4-Round Collision Attack. As shown in Fig. 5, the idea of this attack is to cancel the differences in 8 bytes in each iteration. The probability that 8 bytes match is 2^{-64}, and so 2^{64} pairs following the truncated differential trail for a given input difference have to be generated. The 4-round attack then works as follows:

1. Choose arbitrary message blocks m_1, m_1^* such that $\hat{h}_1$ is fully active.
2. Use a right pair of message blocks m_2, m_2^* for the trail of (3) to cancel 8 bytes of the difference in the state $\hat{h}_2$ (see Fig. 5).

3.–9. Repeat step 2 for message blocks m_i, m_i^*, $3 \leq i \leq 9$, with rotated variants of the trail to cancel another 8 bytes each (see Fig. 5), so that we finally get a full collision in $\hat{h}_9$.

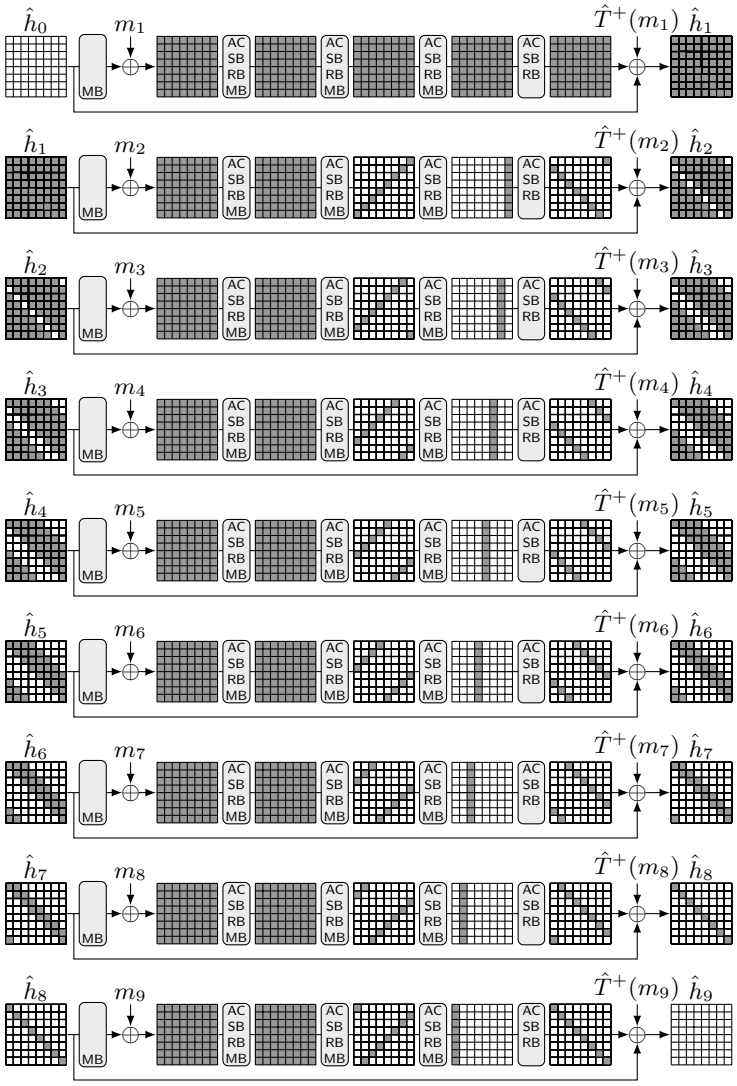

Fig. 5. Hash collision for 4 rounds.

With the help of the rebound attack, the construction of a right pair for one $\hat{T}^{\oplus}$ has a complexity of 2^{64}. The differences have to be canceled iteratively 8 times starting from $\hat{h}_2$ to $\hat{h}_9$. Thus, the overall attack complexity for the collision for 4-round Kupyna-256 is 8 times constructing one right pair, which is $8 \cdot 2^{64} = 2^{67}$.

5-Round Collision Attack. To extend the attack to 5 rounds, we use the truncated differential trail in (4). However, for this trail, we can construct only 2^8 pairs following the trail, and thus each step of the attack on 5 rounds succeeds only with a probability of 2^{-56}. Luckily, this can be compensated by using more message blocks in each step of the attack, as already pointed out in [10]. To cancel 8 bytes of differences in one step, 2^{56} additional blocks (new starting points) are needed. This leads to an attack complexity of $8 \cdot 2^{64+56} = 2^{123}$ and a length of $8 \cdot 2^{56} = 2^{59}$ blocks for the colliding message. However, as discussed in [10], the length of the colliding message pair can be significantly reduced again to 65 message blocks, by using a tree-based approach. Furthermore, the complexity of the attack can be slightly reduced to 2^{120}.

5 Conclusion

In this work, we evaluated the security of Kupyna-256 against rebound attacks. Based on rebound attacks on Grøstl, we mounted collision attacks for up to 5 rounds of the Kupyna-256 hash function and for up to 7 rounds of the Kupyna-256 compression function. This was possible despite the presence of modular constant additions in one of the permutations. These constant additions break the strong alignment of differential trails, leading to more confusion in the propagation of differences. Nevertheless, we were able to adapt the inbound phase of the rebound attack to create a sufficient amount of valid pairs to perform a collision attack. Surprisingly, our results show that the modular constant addition in one permutation does not provide much additional security against rebound attacks, it just complicates the analysis. In the case of Kupyna, the unfortunate choice of round constants for modular addition further decreases the effectivity of the additions. Combined with the lack of other countermeasures (such as different rotation constants), this makes Kupyna an easier target for rebound attacks than the otherwise similar Grøstl hash function. Moreover, the weak alignment of differential trails introduced by the modular constant addition makes it more complicated to bound the minimum number of active S-boxes and might introduce new attack paths. This analysis shows that modular additions inside the permutation are not an optimal choice to diversify similar building blocks, and introduce new problems of their own.

Acknowledgments

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR).
Furthermore, this work has been supported in part by the Austrian Science Fund (project P26494-N15).

A Observation on the Branch Number

Kupyna was clearly designed with the wide-trail strategy in mind. It features the classic AES-like construction with the linear MixBytes function that multiplies each column of the state with an MDS matrix. This function has a differential branch number of 9, meaning that any input column with $a > 0$ active bytes is mapped to an output with at least $9 - a$ active bytes. Though the Kupyna specification features no proofs, this property is usually used to derive bounds on the minimum number of active S-boxes for the primitive.

The other linear functions of each round – in particular the constant and/or subkey xor-addition – do not change this property in AES-like designs. With the modular addition of Kupyna, however, this is no longer true. In particular, this modular addition can lead to carry propagation across byte borders, which also allows propagation of differences across byte borders. This means that the number of active S-boxes over 2 rounds is no longer lower-bounded by 9.

Consider the following example, illustrated in Fig. 6. We investigate the number of active S-boxes over 2 rounds of T^+. For simplicity, we only state the value pair (x_1, x_2) for the first column of state S_1^{RB} through S_2^{AC}; all other columns have zero difference (and RotateBytes does not change the number of active S-boxes). Using the MixBytes matrix $M \in \mathbb{F}_{256}^{8\times 8}$ and AddConstant constant $\zeta_0^{(0)} \in \mathbb{Z}_{2^{64}}$,

$$M = \begin{pmatrix} \texttt{01 01 05 01 08 06 07 04} \\ \texttt{04 01 01 05 01 08 06 07} \\ \texttt{07 04 01 01 05 01 08 06} \\ \texttt{06 07 04 01 01 05 01 08} \\ \texttt{08 06 07 04 01 01 05 01} \\ \texttt{01 08 06 07 04 01 01 05} \\ \texttt{05 01 08 06 07 04 01 01} \\ \texttt{01 05 01 08 06 07 04 01} \end{pmatrix}, \qquad \zeta_0^{(1)} = \begin{pmatrix} \texttt{F3} \\ \texttt{F0} \\ \texttt{F0} \\ \texttt{F0} \\ \texttt{F0} \\ \texttt{F0} \\ \texttt{F0} \\ \texttt{70} \end{pmatrix},$$

we get

$$x_1\colon (\texttt{00 00 00 00 00 00 00 00})^\top \xmapsto{\mathsf{MB}} (\texttt{00 00 00 00 00 00 00 00})^\top \xmapsto{\mathsf{AC}} (\texttt{F3 F0 F0 F0 F0 F0 F0 70})^\top,$$
$$x_2\colon (\texttt{00 00 00 00 00 00 00 FF})^\top \xmapsto{\mathsf{MB}} (\texttt{DB C7 38 AB FF 24 FF FF})^\top \xmapsto{\mathsf{AC}} (\texttt{CE B8 29 9C F0 15 F0 70})^\top,$$
$$\Delta\colon (\texttt{00 00 00 00 00 00 00 FF})^\top \xmapsto{\mathsf{MB}} (\texttt{DB C7 38 AB FF 24 FF FF})^\top \xmapsto{\mathsf{AC}} (\texttt{3D 48 D9 6C 00 E5 00 00})^\top.$$

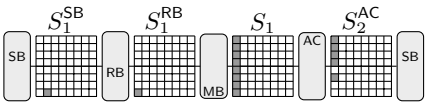

Fig. 6. Example with 6 instead of 9 active S-boxes over 2 rounds of T^+.

References

1. Daemen, J., Rijmen, V.: The wide trail design strategy. In: Honary, B. (ed.) Cryptography and Coding – IMA 2001. LNCS, vol. 2260, pp. 222–238. Springer (2001)
2. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (January 2009), available online: `http://www.groestl.info`
3. Gilbert, H., Peyrin, T.: Super-Sbox cryptanalysis: Improved attacks for AES-like permutations. In: Hong, S., Iwata, T. (eds.) Fast Software Encryption – FSE 2010. LNCS, vol. 6147, pp. 365–383. Springer (2010)
4. Jean, J., Naya-Plasencia, M., Peyrin, T.: Improved rebound attack on the finalist Grøstl. In: Canteaut, A. (ed.) Fast Software Encryption – FSE 2012. LNCS, vol. 7549, pp. 110–126. Springer (2012)
5. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound distinguishers: Results on the full Whirlpool compression function. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. LNCS, vol. 5912, pp. 126–143. Springer (2009)
6. Lamberger, M., Mendel, F., Schläffer, M., Rechberger, C., Rijmen, V.: The rebound attack and subspace distinguishers: Application to Whirlpool. J. Cryptology 28(2), 257–296 (2015)
7. Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., Szmidt, J.: Cryptanalysis of the GOST hash function. In: Wagner, D. (ed.) Advances in Cryptology – CRYPTO 2008. LNCS, vol. 5157, pp. 162–178. Springer (2008)
8. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The rebound attack: Cryptanalysis of reduced Whirlpool and Grøstl. In: Dunkelman, O. (ed.) Fast Software Encryption – FSE 2009. LNCS, vol. 5665, pp. 260–276. Springer (2009)
9. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Rebound attacks on the reduced Grøstl hash function. In: Pieprzyk, J. (ed.) Topics in Cryptology – CT-RSA 2010. LNCS, vol. 5985, pp. 350–365. Springer (2010)
10. Mendel, F., Rijmen, V., Schläffer, M.: Collision attack on 5 rounds of Grøstl. In: Cid, C., Rechberger, C. (eds.) Fast Software Encryption – FSE 2014. LNCS, vol. 8540, pp. 509–521. Springer (2014)
11. Oliynykov, R., Gorbenko, I., Kazymyrov, O., Ruzhentsev, V., Kuznetsov, O., Gorbenko, Y., Boiko, A., Dyrda, O., Dolgov, V., Pushkaryov, A.: A new standard of Ukraine: The Kupyna hash function. Cryptology ePrint Archive, Report 2015/885 (2015), `http://eprint.iacr.org/2015/885`
12. Wu, S., Feng, D., Wu, W., Guo, J., Dong, L., Zou, J.: (pseudo) preimage attack on round-reduced Grøstl hash function and others. In: Canteaut, A. (ed.) Fast Software Encryption – FSE 2012. LNCS, vol. 7549, pp. 127–145. Springer (2012)
13. Zou, J., Dong, L.: Cryptanalysis of the round-reduced Kupyna hash function. Cryptology ePrint Archive, Report 2015/959 (2015), `http://eprint.iacr.org/2015/959`