New techniques for trail bounds and application to differential trails in Keccak

Silvia Mella[1,2], Joan Daemen[1,3] and Gilles Van Assche[1]

[1] STMicroelectronics
[2] University of Milan
[3] Radboud University

Abstract. We present new techniques to efficiently scan the space of high-probability differential trails in bit-oriented ciphers. Differential trails consist in sequences of state patterns that we represent as ordered lists of basic components in order to arrange them in a tree. The task of generating trails with probability above some threshold starts with the traversal of the tree. Our choice of basic components allows us to efficiently prune the tree based on the fact that we can tightly bound the probability of all descendants for any node. Then we extend the state patterns resulting from the tree traversal into longer trails using similar bounding techniques.
We apply these techniques to the 4 largest Keccak-f permutations, for which we are able to scan the space of trails with weight per round of 15. This space is orders of magnitude larger than previously best result published on Keccak-f[1600] that reached 12, which in turn is orders of magnitude larger than any published results achieved with standard tools, that reached at most 9. As a result we provide new and improved bounds for the minimum weight of differential trails on 3, 4, 5 and 6 rounds. We also report on new trails that are, to the best of our knowledge, the ones with the highest known probability.

Keywords: differential cryptanalysis · Keccak · trail weight bounds

1 Introduction

Differential cryptanalysis (DC) exploits predictable difference propagation in iterative cryptographic primitives [BS90]. The basic version makes use of differential trails (also called characteristics or differential paths) that consist of a sequence of differences through the rounds of the primitive. Given such a trail, one can estimate its differential probability (DP), namely, the fraction of all possible input pairs with the initial trail difference that also exhibit all intermediate and final difference when going through the rounds.

The potential of trails in attacks can be characterized by their weight w. For the round function of Keccak-f (but also of, e.g., the AES), the weight equals the number of binary equations that a pair must satisfy to follow the specified differences [BDPV11, DR02]. In particular, a trail is a sequence of round differentials and each round differential imposes a number of conditions on the pair at its input. This number is the weight of that round differential, and the weight of the trail is simply the sum of that of its round differentials. Assuming that these conditions are independent, the weight of the trail relates to its DP as $\mathrm{DP} = 2^{-w}$. In iterated permutations, the existence of low-weight trails over all but a few rounds would imply a low resistance with respect to DC. For estimating the safety margin of a primitive, it is therefore important to understand the distributions of trail weights. One of the goals of this paper is to improve the understanding of differential trails in the Keccak-f permutations.

IACR Transactions on Symmetric Cryptology ISSN 2519-173X, Vol. 2017, No. 1, pp. 329–357
DOI:10.13154/tosc.v2017.i1.329-357

For some primitives there exist compact and powerful proofs lower bounding the weight of trails over multiple rounds. The best known example is the AES with its solid bound of weight 150 over 4 rounds [DR02]. Such compact proofs are out of reach for primitives with weak alignment such as KECCAK-f or ARX-based primitives. For such primitives, the best results can be obtained by computer-assisted proofs, where a program scans the space of all possible n-round trails with weight below some target weight T.

Computer-assisted proofs (and bounds) can be obtained by using standard tools such as (mixed) integer linear programming ((M)ILP) or SAT solvers. MILP has been used by Sun, Hu, Wang, Wang, Qiao, Ma, Shi and Song to prove a lower bound of weight 19 in the first 3 rounds of Serpent, and to find the best 4-round trail in PRESENT that turns out to have weight 12 [SHW+14]. The highest weight per round using standard tools we know of is due to Mouha and Preneel, who used a SAT solver to scan all 3-round characteristics up to weight 26 in the ARX primitive Salsa20 [MP13]. Other examples using (M)ILP do not achieve numerically better results [BFL10, MWGP11].

Better results can be obtained by writing dedicated programs that make use of structural properties of the primitive. For instance, Daemen, Peeters, Van Assche and Rijmen used a dedicated program to scan the full space of trails with less than 24 active S-boxes in Noekeon, and showed that the best 4-round trails have weight 48 [DPVR00]. For KECCAK-f, Daemen and Van Assche used dedicated programs for lower bounding differential trails in KECCAK-f[1600], by generating all 3-round differential trails with weight up to 36, and showed that the best 3-round trails have weight 32 [DV12].

Results obtained with standard tools cited above never reached a weight per round above 9. The dedicated efforts for Noekeon in [DPVR00] and KECCAK-f[1600] in [DV12] both reached a weight per round of 12. Compared to a naive approach (where the space of all n-round trails with weight up to T is covered starting from round differentials with weight up to $\lfloor T/n \rfloor$), the result obtained for KECCAK-f[1600] would be equivalent to investigate trails starting from $2^{48.8}$ round differentials.

Our contributions One of the goals of this work is to extend the space of trails in KECCAK-f that can be scanned with given computation resources. A number of bottlenecks in [DV12] has motivated us to investigate alternative approaches to generate trails more efficiently in order to increase the target weight. Thanks to their abstraction level, some of these techniques are actually more widely applicable than to KECCAK-f. So, we have formalized them in a generic way.

Like in [DV12], we first generate 2-round trails and then extend them in the forward and backward direction. Trails over 2 rounds are generated by constructing state patterns a up to a given cost, where the cost measures the contribution of a to the weight of the trail. We propose a representation of state patterns by elementary components called *units* so that patterns are incrementally generated by adding units. We select cost bounding functions that allow to efficiently bound the cost of any state pattern obtained by the addition of new units. By imposing an order relation on units, a state pattern can be coded as an ordered list of units. This arranges all possible state patterns in a tree, where the parent of a node is the state pattern with the last unit of its list removed. Generating all state patterns with cost up to some budget can be achieved by traversing this tree and pruning when the cost bounding function of a state pattern exceeds the budget.

Thanks to its abstraction level, the tree traversal approach can be applied to primitives with round functions consisting of a bit-oriented mixing layer, a bit transposition layer and relatively lightweight non-linear layer. Of course, the specific definition of units, order relations and cost bounding functions depends on the targeted primitive and its properties.

We finally introduce a number of new techniques to improve the forward and backward extension of trails in KECCAK-f. These techniques exploit the properties of the mixing layer to reduce the number of patterns to investigate and thus the computational effort.

We apply our techniques to scan the space of all 3-round differential trails with weight up to 45 for Keccak-f with widths of 200 to 1600, thereby also covering CAESAR candidates Ketje and Keyak [BDP+14, BDP+15]. As this gives a weight of 15 per round for Keccak-f[1600] doing this the naive way would imply investigating $2^{60.7}$ round differentials. From this perspective, our new techniques allow us to cover a space that is a factor $2^{60.7}/2^{48.8} \approx 4000$ times larger than in [DV12].

The most direct consequence is an extension and an improvement over known bounds for differential trails in Keccak-f as summarized in Table 1. Additionally, scanning a bigger space of trails for different variants of Keccak-f allows to compare results and trends among widths and it gives insight in how weight distributions of low-weight trails scale with the permutation width of Keccak-f. As a by-product of the trail search, we also report on new trails with the lowest known weight (Table 3).

Table 1: Minimum weight or range for the minimum weight of trails for different variants of Keccak-f per instance of Keccak-f[b] and number of rounds. A range $[x, y]$ means that x is a lower bound and y is the lowest known weight.

rounds	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
2	8 [BDPV11]	8 [BDPV11]	8 [BDPV11]	8 [BDPV11]
3	20 [BDPV11]	24 [this work]	32 [this work]	32 [DV12]
4	46 [BDPV11]	[48,63] [this work]	[48,104] [this work]	[48,134] [this work]
5	[50,89] [this work]	[50,147] [this work]	[50,247] [this work]	[50,372] [this work]
6	[92,142] [this work]	[92,278] [this work]	[92,556] [this work]	[92,1112] [this work]

Paper structure The paper is structured as follows. In Section 2 we recall some basics about differential trails and how to scan the space of trails up to a given weight. In Section 3 we present our strategy to generate differential patterns as a tree traversal and in Sections 4, 5 and 6 we show how to cover the space of 6-round trails in Keccak-f. Finally, in Section 7 we provide experimental results. For the interested reader, in the Appendix we provide more details about experimental results, as the number of trails generated and the execution time. Finally, we compare our method with previous work.

2 Scanning the space of trails

In this Section we recall some facts about differential trails and how to cover the space of trails up to a given weight.

2.1 Trails and trails extension

For a transformation f operating on $\mathbb{Z}_2^N$, a differential pattern is an element of $\mathrm{GF}(2)^N$ representing the difference between two states $a_1 \oplus a_2$. A bit in a differential pattern is active if it has value 1 and passive otherwise.

The differential probability of a differential (u', v') is defined as

$$DP(u' \xrightarrow{f} v') = \frac{\#\{u \ : \ f(u) \oplus f(u \oplus u') = v'\}}{\#\{u\}}$$

and the weight is defined as the negative logarithm of its probability [DR01].

A trail over n rounds of an iterative permutation is specified by the input difference, the output difference and the intermediate differences, i.e. the differences after each round. For a primitive f with linear layer λ and non-linear layer χ, we specify differential trails

by the differences before and after each layer of each round:

$$Q = a_1 \xrightarrow{\lambda} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda} \quad \dots \quad \xrightarrow{\chi} a_{n+1}$$

and we denote it as $Q = (a_1, b_1, \dots, a_{n+1})$.

The weight of a trail is the sum of the weight of the round differentials that compose the trail. Since λ is linear and $b_i = \lambda(a_i)$, the weight can be expressed as $\mathrm{w}(Q) = \sum_i \mathrm{w}(b_i \xrightarrow{\chi} a_{i+1})$.

Since χ is not linear, given a sequence of patterns $\tilde{Q} = (a_1, b_1, a_2, b_2, \dots, b_n)$ there are several n-round trails with $\tilde{Q}$ as leading part, which differ from each other for the pattern a_{n+1}. The sequence $\tilde{Q}$ is called a *n-round trail prefix*.

Given a n-round trail prefix, it is possible to extend it to $n+1$ rounds. The extension can be done in both the forward and backward direction. Forward extension consists in building all differences a_{n+1} that are compatible with b_n through χ and then compute the corresponding $b_{n+1} = \lambda(a_{n+1})$.

Backward extension consists in building all differences b_0 compatible with a_1 through χ^{-1}. The minimum weight over all these compatible differences b_0 is called the *minimum reverse weight* and denoted by $\mathrm{w}^{\mathrm{rev}}(a_1)$. Hence a trail prefix $\tilde{Q}$ defines a set of trails with $\tilde{Q}$ as trailing part, with b_0 compatible with a_1 through χ^{-1} and with weight at least $\mathrm{w}(\tilde{Q}) + \mathrm{w}^{\mathrm{rev}}(a_1)$. This set of trails is called *trail core* [DV12]. Hence the search of trails can be limited to such trail cores.

In many round functions there is a lot of symmetry and this symmetry is only broken by round constants or round keys. However, the weight of trails is independent of round constants or keys and hence it may be the case that there are large classes of equivalent trails. Therefore it is sufficient to generate one representative trail for class.

2.2 The naive method: trails from extending round differentials

An n-round trail with weight W has at least one round differential with weight below or equal to $L = \lfloor \frac{W}{n} \rfloor$. So we can find all n-round trails with weight W with a first-order approach that consists of the following two phases:

1. Generate all round differentials of weight below or equal to L.
2. Build for each of these round differentials all n-round trails that contain them by:
   - extending forward by $n-1$ rounds,
   - extending backward by 1 round and forward by $n-2$ rounds,
   - ...
   - extending backward by $n-2$ rounds and forward by 1 rounds,
   - extending backward by $n-1$ rounds.

The problem of this method is that the number of round differentials increases very steeply with the weight and the value of W we can achieve is limited by the mere quantity of round differentials with weight below L.

2.3 The second-order method: trails from extending two-round trails

In modern ciphers the round function has a relatively powerful mixing layer and the number of two-round trails with weight below $2L$ is much smaller than the number of round differentials with weight below L.

For example, AES has $16 \times 255 \times 127 + \binom{16}{2} \times 255^2 \times 127^2 > 10^{11}$ round differentials with weight below 15 while the mixing layer ensures that there are no two-round trails

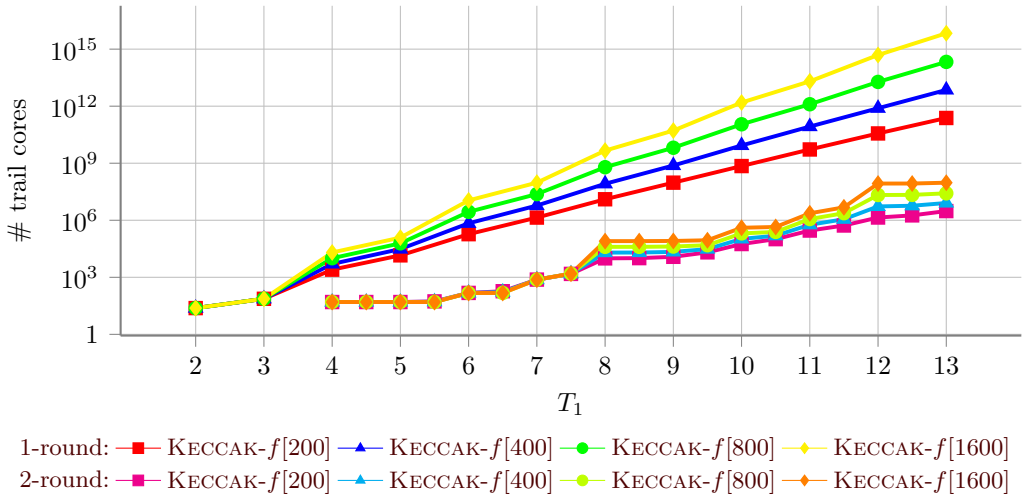


Figure 1: Number of 1-round and 2-round trail cores in Keccak-f modulo z-symmetry with weight T_1 per round.

with weight below 30. In fact, the mixing layer and the byte-aligned layout of AES allows demonstrating that there are no four-round trails with weight below 150 without the need for scanning the space. The price paid for this is a relatively costly mixing layer and the vulnerability of the cipher with respect to attacks that exploit the strong alignment. However, most weakly aligned ciphers and permutations have a mixing layer that has a similar effect.

We depict in Figure 1 the number of round differentials and 2-round trails of Keccak-f for several widths, already taking into account the symmetry along z (and hence reducing the search space by a factor equal to the length of the lane, see Definition 1). As can be seen, the number of 1-round trails below weight L is much bigger than the number of 2-round trails below $2L$.

Instead of building trails from round differentials, we can build them from two-round trails. For n even, an n-round trail of weight W always contains a two-round trail of weight below or equal to $2L+1$ with $L = \lfloor \frac{W}{n} \rfloor$. So now the two phases become:

1. Generate all two-round trails of weight below or equal to $2L$.
2. Build for each of these round differentials all n-round trails that contain them by forward and backward extension.

This allows to target much higher values of W for the same number of rounds n. For this reason, this approach was used in [DV12] and we will use it, too.

3 Generating two-round trail cores as a tree traversal

The weight of a 2-round trail core (a, b) is fully determined by a (or $b = \lambda(a)$). We call the function that gives the weight of the two-round trail core as a function of a, the *cost* of a. It now suffices to generate all patterns a with cost below $2L$.

To do this in an efficient way, we introduce a new approach where we arrange all possible patterns a in a tree, where the root of the tree is the all-zero (or empty) pattern. In its simplest and most intuitive form, the idea is to consider descendants by monotonically adding active bits both in a and in b, in a way that depends on the properties of λ. Monotonicity is actually not required, yet what we need is a lower bound on the cost of a

pattern and of all its descendants. Finding all patterns with cost below $2L$ now simply corresponds to traverse the tree and prune any node (and all its descendants) whose lower bound on the cost is higher than $2L$.

3.1 Units and unit-lists

To exhaustively scan all the possible patterns up to a given budget W, we represent a pattern as a set of elements called *units*. These units are groups of active bits with particular properties. State patterns are hence incrementally generated by incrementally adding units. This arranges all patterns in a tree, where the children of a node are obtained by adding a new unit to it.

When building patterns, we do not want to generate them more than once. For example, we want to avoid generating a pattern with units s_1 and s_2 by first adding s_1 then s_2 and then again by first adding s_2 then s_1. A straightforward way to implement it is to define a total ordering $\prec$ on units.

Equipped with this ordering, a pattern can be represented as an ordered list of units, called *unit-list*, $s = (s_i)_{i=1,\dots,n}$ that satisfies $s_1 \prec s_2 \prec \cdots \prec s_n$. Namely, the children of a node s are $s \cup \{s_{n+1}\}$ for all s_{n+1} such that $s_n \prec s_{n+1}$. Equivalently, the *parent* of a pattern is defined as the pattern with its last unit removed. This entirely defines the tree structure: the root is the empty state and the list of units of a node defines the path from the node to the root. Traversing the tree simply consists of recursively generating the children of a given node by iterating over all units that come after the last unit of that node.

To efficiently traverse the tree, we define a function that gives a lower bound on the cost of a node and all its descendants. Let γ denote the cost function of a pattern. Let the set of descendants of s, including s itself, be denoted by $\triangle(s)$. Let $L(s)$ be a lower bound on the cost of the elements of $\triangle(s)$, that is, $L(s)$ satisfies $\gamma(s') \geq L(s)$ for all $s' \in \triangle(s)$. Then, any subtree starting from s can be safely ignored when $L(s) > W$. In other words, as soon as the search encounters a pattern s such that $L(s) > W$, s and all its descendants can be ignored.

In the tree traversal, the move from a node $s = (s_i)_{i=1,\dots,n}$ to the next one is defined following Algorithm 1. In particular, if possible we try to add a new unit s_{n+1} after s_n. This is done by the function `toChild()`, which returns true if a new unit has been added and false otherwise. If it is not possible, either because of the budget or because there are no units left, we iterate the value of unit s_n. This is done by function `toSibling()`, which returns true if a new valid value for s_n is found, false otherwise. If neither this action is possible, we remove s_n from the unit-list (using function `toParent()`) and iterate s_{n-1}. And so on. The search ends when there are no more units left in the unit-list. This check is performed by the function `toParent()`, which returns true if the last unit has been popped from the unit-list, false if the unit-list is empty.

This method of generating state patterns as a tree traversal can be applied to all round functions that have a linear and a non-linear layer. However, the definition of units and lower bound functions depends on the details of these layers and thus it is specific of the target cryptographic primitive. We will define them for Keccak-f in the next sections.

3.2 Searching by canonicity

The tree traversal approach can be extended when there are symmetry properties that we wish to take into account. We use these symmetry properties to reduce our effort, by visiting only those nodes that are the representative of their equivalence class. To this end, we define a property that we call *canonicity*: a pattern is canonical if it is the *smallest* representative of its class.

Algorithm 1 Next move in the tree traversal

```
1: if toChild() then                          ▷ adds a new unit to the list
2:     return true
3: do
4:     if toSibling() then                    ▷ iterates the highest unit of the list
5:         return true
6:     if !toParent() then                    ▷ pops the highest unit of the list
7:         return false
8: while (true)
```

At first, we need to define a total order relation among patterns. Let $T = \{\tau\}$ be a set of transformations s.t. f is symmetric with respect to each $\tau \in T$ (i.e., $f \circ \tau = \tau \circ f$). Given a total order relation $s \prec s'$ among patterns, we say that s is *canonical* if it is the minimum among all the transformed variants of s, i.e., the smallest one in its equivalence class:

$$s \text{ is canonical iff } s = \min_{\prec} \{\tau(s) \ : \ \tau \in T\}\,.$$

We use for this the unit list: the order of the units naturally establishes an order of unit lists and the representative with the smallest unit list is the canonical one. For two patterns s and s' represented as unit-lists $s = (s_i)_{i=1,\ldots,n}$ and $s' = (s'_j)_{j=1,\ldots,m}$, we define the lexicographic order relation $s \prec_{\text{lex}} s'$ using the relation $\prec$ on their units as

$$s \prec_{\text{lex}} s' \text{ iff } \begin{cases} \exists\ k \text{ such that } s_i = s'_i\ \forall\, i < k \text{ and } s_k \prec s'_k, \text{ or} \\ n < m \text{ and } s_i = s'_i\ \forall\, i \leq n. \end{cases} \tag{1}$$

In other words, either s' and s are in two different branches of the tree with the same ancestor at height $k-1$ and $s_k \prec s'_k$, or s' is a descendant of s.

The following lemma gives an interesting property of this ordering in the context of canonicity.

Lemma 1. *Using the lexicographic order, the parent of a canonical pattern is canonical.*

Proof. Let s be a canonical pattern. Then for each $s' \neq s$ in the equivalence class of s, Eq. (1) holds. The parent of s is s with its highest unit s_n removed. In $s' = \tau(s) = (s'_i)_{i=1,\ldots,n}$, $s'_{\Pi(n)}$ is not necessarily the highest unit. Say that $\Pi(n) = j$, so that $\tau(\text{parent}(s)) = (s'_1, \ldots, s'_{j-1}, s'_{j+1}, \ldots, s'_n)$. Then, there are two possible cases:

- $\diamond$ $j > k$: then the right hand of (1) still holds;
- $\diamond$ $j \leq k$: then in (1) s'_j is replaced by s'_{j+1}, which is higher than s'_j, and we have $s_i = s'_i\ \forall\, i < j$ and $s_j \prec s'_{j+1}$.

In all cases, the parent of s is canonical. □

A direct consequence of Lemma 1 is that a pattern that is not canonical cannot have a canonical descendant. So when we encounter non canonical patterns during our tree traversal, we can safely exclude them and all their descendants.

Therefore, there are two ways to prune entire subtrees. One when we encounter a node that is not canonical and the other when we encounter a node whose cost (and that of its descendants) is above the budget.

4 Scanning the space of trails in Keccak

We apply the approach described in previous sections to scan the space of 6-round trail cores in KECCAK-f up to a given weight. We start from 3-round trail cores and extend them by 3 rounds in the backward and forward direction. Three-round trail cores are in turn generated by extending 2-round trail cores. To generate them, we define units and cost-bounding functions to apply the tree-traversal method. Then we perform optimized extensions based on the properties of the mixing layer of KECCAK-f.

4.1 Keccak-f

KECCAK-f is a family of seven permutations denoted by KECCAK-$f[b]$ [BDPV11], with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. The state of KECCAK-$f$ is organized as an array of $5 \times 5 \times w$ elements of GF(2) called bits, with $w \in \{1, 2, 4, 8, 16, 32, 64\}$. A bit of the state is specified by (x, y, z) coordinates considered modulo 5 for x and y and modulo w for z. A *row* is a set of 5 bits with given (y, z) coordinates, a *column* is a set of 5 bits with given (x, z) coordinates and a *slice* is a set of 25 bits with given z coordinate.

Each permutation consists in the repetition of a round function, which is composed of five invertible steps: θ is the linear mixing layer; ρ is a bit-wise cyclic shift operation within the lanes with different offsets; π is a transposition of lanes; χ is a non-linear mapping of algebraic degree 2; ι adds a round constant to lane $(0, 0)$.

Since ι has no effect on the compatibility of differential patterns, the linear layer reduces to $\lambda = \pi \circ \rho \circ \theta$ when working with differentials. When we refer to state patterns, we always imply it to be at the input of θ, unless explicitly stated otherwise. When a property is verified at the input of θ we use the terminology *at* a. Similarly, we speak of *after* θ and *at* b.

All the steps of KECCAK-f involved in difference propagation, i.e. λ and χ, are invariant with respect to translation along z-axis [BDPV11]. When generating trails we must take into account this symmetry property and hence consider only so called *z-canonical* trails.

Definition 1. A pattern s is *z-canonical* if and only if

$$s = \min_{\prec} \{\tau_z(s) \ : \ z \in \mathbb{N}\},$$

where $\tau_z(s)$ is the pattern s translated by z positions along the z axis.

The weight of a differential (b_i, a_{i+1}) in KECCAK-f depends only on b_i [DV12]. Hence all n-round trails Q that share the sequence of patterns $\tilde{Q} = (a_1, b_1, a_2, b_2, \ldots, b_n)$ and with a_{n+1} compatible with b_n have the same weight and this depends only on b_i.

4.1.1 Properties of θ

Since our definition of units for KECCAK-f is determined by the properties of θ, we briefly recall them here.

Given a state pattern a, its *parity* $p(a)$ is defined by $p[x, z] = \sum_y a[x, y, z]$. A column $[x, z]$ is *even* if $p[x, z] = 0$, otherwise it is *odd*. If all columns of a are even we say that a is in the *(parity) kernel*, otherwise we say it is outside the (parity) kernel [BDPV11]. We denote the set of states in the kernel with $|K|$ and the set of states outside the kernel with $|N|$, where the vertical lines symbolize the χ steps.

The θ map adds a pattern, that depends on the parity, to the state. This pattern is called θ-*effect* and is defined by $E[x, z] = p[x - 1, z] + p[x + 1, z - 1]$. Hence $\theta(a)[x, y, z] = a[x, y, z] + E[x, z]$. A column (x, z) is called *affected* if $E(x, z) = 1$, otherwise it is *unaffected*. We thus distinguish four different types of columns: unaffected even, unaffected odd, affected even and affected odd. A state in the kernel has only unaffected even columns.

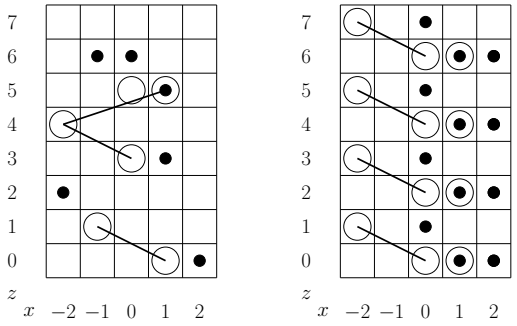


Figure 2: Example of parity patterns and θ-effect superimposed. A circle represents an odd column, a dot represents an affected column. The odd columns of a run are connected through a line. There are three runs in the first pattern: one of length 1 starting at $(0, 5)$, one of length 2 starting at $(1, 0)$ and one of length 3 starting at $(0, 3)$. Note that $(1, 5)$ is an odd column belonging to the run of length 3 and affected by the run of length 1. The second is an example of a symmetric parity pattern with period 2.

Given a parity pattern, a *parity-bare* state is a state with the minimum possible number of active bits at a and b over all states with the given parity. Such a state has zero active bits in unaffected even columns and one active bit in each unaffected odd column, while each affected column has 5 active bits, where each bit appears either at a or at b [BDPV11].

Given a parity pattern, we can group its odd columns into *runs* [BDPV11]. Two odd columns belong to the same run if their (x, z) coordinates differ by $(-2, 1)$. A run is fully specified by its starting point (x_0, z_0) and its length ℓ. Each run contains ℓ odd columns and affects two columns:

- starting affected column: the right neighbor of its initial point at $(x_0 + 1, z_0)$ and
- ending affected column: the top-left neighbor of its final point at $(x_0 + 1 - 2\ell, z_0 + \ell)$.

An odd column of a run might be affected by another run. Figure 2 gives two examples of parity patterns as a set of runs indicating their odd columns and the affected columns. A parity pattern is symmetric if there exists an integer $\bar{z}$ such that $\tau_{\bar{z}}(s) = s$. The integer $\bar{z}$ is called the *period* of s.

4.2 Scanning the space of 6-round trails

We can generate all 6-round trail cores up to weight $2T_3 + 1$ by doing forward extension by 3 rounds, or backward extension by 3 rounds, of all 3-round trail cores with weight up to T_3. Indeed, any such 6-round trail satisfies either $\mathrm{w}(b_0) + \mathrm{w}(b_1) + \mathrm{w}(b_2) \leq T_3$ or $\mathrm{w}(b_3) + \mathrm{w}(b_4) + \mathrm{w}(b_5) \leq T_3$. If a trail of weight $2T_3 + 1$ or less is found, then it yields a lower bound on 6-round trail weights. Otherwise, the bound is $2T_3 + 2$, not necessarily tight.

Scanning the space of 3-round trail cores up to weight T_3 gives bounds also on 4 and 5 rounds. Indeed, any 4-round trail satisfies either $\mathrm{w}(b_0) + \mathrm{w}(b_1) + \mathrm{w}(b_2) \leq T_3$ or $\mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) \leq T_3$ and the weight over a single round is at least 2. If no trail is found below the limit, then we can say that the lower bound on the weight of 4-round trail cores is $T_3 + 3$. Any 5-round trail may satisfy the condition $\mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) \leq T_3$ and $\mathrm{w}(b_0) = \mathrm{w}(b_4) = 2$. If no trail core with such a weight profile is found, we can say that a lower bound on the weight of 5-round trail cores is $T_3 + 5$.

4.3 Scanning the space of 3-round trail cores

To scan the space of 3-round trail cores (a_1, b_1, a_2, b_2) up to some limit weight T_3, we distinguish between different sets depending on a_1 and a_2 being in the kernel. In fact, for any given limit, the number of trail cores in $|K|$ is much higher than those in $|N|$. This calls for different approaches and it is reflected in our strategy.

Thus there are four classes: $|K|K|$, $|K|N|$, $|N|K|$ and $|N|N|$. Any 3-round trail belongs to one and only one of these classes. For brevity we denote the weights of the three round differentials by shortcut notations: $\mathrm{w}^{\mathrm{rev}}(a_1) = w_0$, $\mathrm{w}(b_1) = w_1$ and $\mathrm{w}(b_2) = w_2$.

To cover the different sets, we extend 2-round trail cores (a, b) by one round in the forward or backward direction. Extension is limited to compatible patterns in the kernel or outside the kernel, depending on the set we are targeting.

For $|K|K|$ we use the method of [DV12, Section 7]. This consists in generating all 2-round trail cores with a in the kernel for which there exists a compatible state $\chi(b)$ in the kernel. Then, forward extension (in the kernel) of these trails is performed.

For $|K|N|$, we partition the space based on the value of w_0.

Lemma 2. *All trail cores in $|K|N|$ up to weight T_3 can be generated by forward extending by one round outside the kernel all trail cores in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) \leq T_1$ and backward extending by one round in the kernel all trail cores in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$.*

Proof. For a given T_1, any trail core Q in $|K|N|$ satisfies either $w_0 \leq T_1$ or $w_0 > T_1$.

If $w_0 \leq T_1$, then the trail Q starts with a two-round trail core in $|K|$ with $\mathrm{w}^{\mathrm{rev}}(a) \leq T_1$ and hence can be built by forward extending this by one round outside the kernel.

If $w_0 > T_1$, then $w_1 + w_2 < T_3 - T_1$. So, the trail core Q ends with a two-round trail core in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$ and hence can be built by backward extending this by one round in the kernel. □

The optimal value of T_1 is determined by the ratio of 2-round trails in $|K|$ and $|N|$ for the relevant weight functions and the cost of extension.

For $|N|K|$, we apply the same partitioning technique, but based on the weight of w_2.

Lemma 3. *All trail cores in $|N|K|$ up to weight T_3 can be generated by backward extending by one round outside the kernel all trail cores in $|K|$ with $\mathrm{w}(b) \leq T_1$ and forward extending by one round in the kernel all trail cores in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$.*

Proof. For a given T_1, any trail core Q in $|N|K|$ satisfies either $w_2 \leq T_1$ or $w_2 > T_1$.

If $w_2 \leq T_1$, then the trail Q ends with a two-round trail core in $|K|$ with $\mathrm{w}(b) \leq T_1$ and hence can be built by backward extending this by one round outside the kernel.

If $w_2 > T_1$, then $w_0 + w_1 < T_3 - T_1$. So, the trail core Q starts with a two-round trail core in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3 - T_1$ and hence can be built by forward extending this by one round in the kernel. □

Lemma 4. *All trail cores in $|N|N|$ up to weight T_3 can be generated by forward extending by one round outside the kernel all trail cores in $|N|$ with $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) < T_3$ and backward extending by one round outside the kernel all trail cores in $|N|$ with $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq T_3$.*

Proof. Any trail core Q in $|N|N|$ satisfies either $w_2 > w_0$ or $w_2 \leq w_0$.

If $w_2 > w_0$, combination with $w_0 + w_1 + w_2 \leq T_3$ gives $2w_0 + w_1 < T_3$. It follows that all trail cores Q in $|N|N|$ with $w_2 > w_0$ start with a two-round trail core in $|N|$ with $2w_0 + w_1 < T_3$ and hence can be built by forward extending these by one round outside the kernel.

If $w_2 \leq w_0$, combination with $w_0 + w_1 + w_2 \leq T_3$ gives $w_1 + 2w_2 \leq T_3$. It follows that all trail cores Q in $|N|N|$ with $w_2 \leq w_0$ end with a two-round trail core in $|N|$ with $w_1 + 2w_2 \leq T_3$ and hence can be built by backward extending these by one round outside the kernel. □

5 Generating two-round trail cores in Keccak-f

We generate 2-round trail cores applying the tree-traversal method described in Section 3. In order to do this, we need to define units, order relations among units and cost-bounding functions.

5.1 Generating trail cores in the kernel

In this section, we deal with the generation of 2-round trail cores in the kernel with either $\mathrm{w}^{\mathrm{rev}}(a)$ or $\mathrm{w}(b)$ at most W.

State patterns a in the kernel have an even number of active bits in each column, namely zero, two or four. A pair of active bits in the same column is called an *orbital*. A column can thus have either zero, one or two orbitals.

An orbital is determined by the coordinates of its active bits and we represent it as a 4-tuple $(x, \{y_1, y_2\}, z)$ with the convention that $y_1 < y_2$.

We define an order relation on orbitals as the lexicographic order on $[z, x, y_1, y_2]$. To represent a column containing two orbitals in a unique way, we take the convention that $(x, \{y_1, y_2\}, z)$ and $(x, \{y_1', y_2'\}, z)$ satisfy $y_2 < y_1'$, namely, the first bit of the second orbital is higher than the second bit of the first orbital.

Any state pattern in the kernel can thus be represented as an ordered list of orbitals. This allows us to arrange states in the kernel in a tree where the parent of a node is the state pattern with the last orbital removed. At the root of the tree there is the empty state. Therefore, to concretely build state patterns in the kernel we start from the empty state and iteratively add orbitals after the last orbital already in the state.

To bound the cost of all descendants of a given pattern, we can use the cost of the pattern itself.

Lemma 5. *The cost function of a pattern lower bounds the weight of all its descendants.*

Proof. Given a 2-round trail core (a, b), adding an orbital to a adds two active bits at a and two active bits at b. This is a consequence of the fact that the θ mapping acts as the identity in the kernel, so bits active at a are also active at b but in different positions due to the ρ and π mappings. Hence, whether we consider $\mathrm{w}^{\mathrm{rev}}(a)$ or $\mathrm{w}(b)$, the cost is monotonic with respect to addition of orbitals, because both these weights are monotonic when adding active bits [DV12]. □

From Lemma Lemma 5 it follows that we can avoid to add new orbitals to a pattern when it has weight already above the budget.

Remark on z-canonicity When generating trail cores in the kernel, a z-canonical pattern must have an orbital in the first slice. In fact, for any pattern consider the first orbital in its orbital-list, say $(x, \{y_1, y_2\}, \bar{z})$. If we translate the pattern by $\bar{z}$ positions along the z-axis, we obtain an orbital-list whose first orbital is $(x, \{y_1, y_2\}, 0)$ that is smaller than $(x, \{y_1, y_2\}, \bar{z})$ if $\bar{z} \neq 0$, using the lexicographic order defined above.

5.2 Generating trail cores outside the kernel

In this section, we deal with the generation of 2-round trail cores in $|N|$ with the cost function of the form $\alpha \mathrm{w}^{\mathrm{rev}}(a) + \beta \mathrm{w}(b)$ with $\alpha, \beta \in \{1, 2\}$.

5.2.1 Parity-bare states and orbital trees

Removing an orbital from an unaffected column of a state pattern at a also removes two bits at b. We call an orbital in an unaffected column a *free orbital*. So the removal of a free orbital to a state cannot increase the cost.

We can use parity-bare states and free orbitals as building blocks.

Lemma 6. *Each state can be decomposed in a unique way in a parity-bare state and a list of free orbitals, where the active bits of the parity-bare state do not overlap with those of the orbitals, the active bits of the orbitals do not overlap mutually and the parity-bare state and the original state have the same parity.*

Proof. We describe a recursive procedure for performing the decomposition by removing free orbitals from the state one by one in a unique way.

We start from the state and an empty list of free orbitals. Over all unaffected columns with more than a single active bit, choose the one with the highest (x, z) coordinates, where the order among column positions is the lexicographic order $[z, x]$. Remove the two active bits in this column with the highest y coordinates from the state and add them as a free orbital at the beginning of the list of free orbitals. Repeat this step operation until there are no unaffected columns left in the state with more than a single active bit.

This procedure ends as the number of active bits in a state is finite. The resulting state is parity-bare as any unaffected column has either 0 (even) or 1 (odd) affected bits. That parity-bare state has the same parity as the original state as removing an orbital does not affect the parity. Finally, the active bits of the free orbitals and parity-bare state do not overlap as every free orbital is taken from an unaffected column and hence removes bits at a and at b. □

The list of orbitals defines a tree that we call an *orbital tree*. In this tree the parent of a node is the state with the highest free orbital removed and at its root is a parity-bare state. The tree of state patterns in the kernel is a special case of this: the orbital tree with the empty state at its root. All nodes in a orbital tree have the same parity as adding orbitals does not modify the parity. The children of a node have cost at least that of their parent, as explained in Subsection 5.1.

It remains to explain how to generate parity-bare states.

5.2.2 Column assignments and the run tree

In a parity-bare state an affected column can assume 16 possible values and an unaffected odd column can assume 5 possible values. Fixing the value of a column is called *column assignment*.

For our purposes, it is natural to represent a parity-bare state as a list of column assignments. Furthermore, we group them in runs. For each run r, a state pattern consists of the column assignments of the unaffected odd columns of r and of the two affected even columns of r.

When there is more than one run, the odd column of a run can be affected by another run, resulting in an affected odd column. In such a case, to preserve the grouping into runs, we express the column assignment of an affected odd column as the bitwise sum of two column assignments: the unaffected odd column of one run, and the affected even column of the other run. To make this representation unique, we choose as a convention that the unaffected odd column must be an odd-0 column, where we define an *odd*-0 column as an unaffected odd column with a single active bit in $y = 0$.

We define an order relation over column assignments based on their type and on the associated runs. In a run starting at (x, z), the starting affected column in $(x + 1, z)$ comes before the first odd column in (x, z), which comes before the second odd column in $(x - 2, z + 1)$ and so on until the ending affected column in $(x_0 + 1 - 2\ell, z_0 + \ell)$ which comes at the end. When comparing two columns belonging to two different runs, the one corresponding to the smallest run comes before, where the smallest run is given by the lexicographic order $[z, x, \ell]$. It follows that in the incremental generation of state patterns, a new run is not started until the previous one has been completed.

A parity-bare state pattern can thus be represented as a list of column assignments. This naturally defines a tree containing all parity-bare states that we call the *run tree*. In this tree the parent of a node is the state with the last column assignment removed and its root is the empty state. Clearly, not all nodes of the tree define a valid state pattern. Indeed a state is valid if its last run is complete, i.e. if the last column in the column-list is an ending affected column. Only such patterns are concretely output.

Therefore, building all state patterns outside the kernel can be done by traversing the run tree and then for each parity-bare state traverse its orbital tree.

Bounding the cost Adding an affected even column assignment to a in a column position where there is already an unaffected odd column assignment (or vice versa) is not monotonic in the weight. For that reason we cannot use the cost of a pattern to prune the tree, but must define a lower bound for the cost of all its descendants. It turns out that we can define such a lower bound that is reasonable tight.

By taking into account the worst-case loss due to the addition of overlapping column assignments, we can lower bound the cost of all descendants of a parity-bare state:

1. Adding an odd-0 column assignment to an affected even column moves the active bit at $y = 0$ from a to b or vice versa.

2. Adding an affected even column to an odd-0 column makes either its active bit at a or its active bit at b passive, but never both. It depends on the particular affected even assignment and hence is not known in advance. As a special case, if the bit of the odd-0 column is the only active bits in its slice at a, adding an even column assignment cannot reduce the number of active bits in that slice to 0 as it adds an even number of bits and the contribution of the bit at a remains the same.

Definition 2. The bound on the descendants of a pattern is defined as follows. Given the parity-bare state a and its corresponding state b compute reduced states $\bar{a}$ and $\bar{b}$ as follows. For each affected even column remove the active bit in $y = 0$ at a or in the corresponding position at b. For every odd-0 column, if it is in a slice whose only active row is $y = 0$, then remove the active bit in the corresponding position at b. Let N_0 be the set of unaffected odd-0 columns whose slices at a contain at least one active row different from $y = 0$. The bound on the descendants of a is $L(a) = \alpha \cdot \mathrm{w}^{\mathrm{rev}}(\bar{a}) + \beta \cdot \mathrm{w}(\bar{b}) - 2 \cdot \max(\alpha, \beta) \cdot \#N_0$.

Lemma 7. *The bound $L(a)$ lower bounds the weight of the parity-bare state a and all its descendants.*

A proof of the Lemma above is given in the appendix.

Note on z-canonicity Verifying z-canonicity in the run tree requires comparing lists of column assignments. Similar to the case in the kernel with orbitals, also in this case the first column assignment must be restricted to the first slice.

For state patterns in an orbital tree, verifying z-canonicity is only required if the parity-bare state at its root exhibits symmetry. In that case it just requires comparing lists of orbitals. Note that this is a rare case, except for the orbital tree with zero parity.

6 Extension of trails in Keccak-f

In this section we present techniques for speeding up extension of trails in Keccak-f and so increasing the size of the trail space we are able to scan.

The general principles are similar to those for two-round trail core generation. We make use of the linearity of λ, grouping of state patterns by their parity and we use monotonic weight bounding functions.

6.1 Forward extension

Given a trail core Q ending in pattern b, forward extension up to some weight T by one round can be seen as scanning the space of all state patterns a that are χ-compatible with b and checking whether the weight of the extended trail $(Q, a, \lambda(a))$ is below T.

As shown in [BDPV11], the set of values a that are compatible with b through χ forms an affine space that we denote as $U(b)$ (or just U if b is clear from the context). This affine space U can be described as $U = V + e$ with V a vector space of dimension $\mathrm{w}(b)$ and e a state pattern that forms the offset. Offsets and bases for constructing the affine space are given in [BDPV11]. If b has large weight, brute-force scanning all $a \in U$ becomes prohibitively expensive and therefore we seek for more efficient methods.

First of all, recall from Section 4.3 that we treat extension with a inside the kernel or a outside the kernel separately. The former can be made efficient in a particularly simple way, the latter requires some more sophistication.

6.1.1 Forward extension inside the kernel

The kernel is a vector space and we will denote it by K. The set of state patterns that are χ-compatible with b and are in the kernel is given by $(V + e) \cap K$. The intersection is either empty, or it is an affine space $U' = V' + e'$, with V' the vector space $V' = V \cap K$ and e' an offset in the kernel.

We use the structure of χ to construct an efficient algorithm that, given b, returns an offset $e' \in U(b) \cap K$ or a message stating that $U(b) \cap K$ is empty.

This algorithm operates slice by slice, where it makes use of the fact that χ operates on individual rows and K is computed over individual columns.

In particular, each active row defines an affine space covering all row-patterns that are χ-compatible with it (for the explicit definition of offsets and bases see [DV12, Table 3]). The active rows in a slice $b[z]$ thus define an affine space $U[z]$ covering all slice-patterns that are χ-compatible with $b[z]$. To generate its offset we take an empty slice and add the offsets of all active rows to it. We denote this action by $e[z] = \sum_y e[y, z]$. Then we take a basis vector of an active row and add it to an empty slice. The obtained slice is added to the set of basis vectors for $U[z]$. We repeat the process for all basis vectors of all active rows. We denote this action by $V[z] = \bigcup_y V[y, z]$.

The affine space U is defined by the set of basis vectors of all active slices and an offset that is the sum of offsets of all active slices. Using notation introduced above, we say $e = \sum_z e[z]$ and $V = \bigcup_z V[z]$.

For e to be in the kernel, all $e[z]$ must be in the kernel so this can be handled slice by slice. For all z we can compute parities of offset and of basis vectors. These in turn generate the space of parity patterns that are χ-compatible with b. We denote this space as $U_p = e_p + V_p$.

If $e_p[z] \notin \mathrm{Span}(V_p[z])$, then all elements in $U[z]$ will have at least one odd column and so will not be in the kernel. This is the case for instance when $e[z]$ has a single active row or has two active bits in different columns. Therefore, slices with single active row and slices with two non-orbital active bits allow eliminating in a very efficient way the vast majority of candidates b for extension. A non-trivial example is given in Figure 3.

If $e_p[z] \in \mathrm{Span}(V_p[z])$, then we generate $e'[z]$ by adding to $e[z]$ a number of basis vectors such that the sum of their parities is equal to $e_p[z]$. A non-trivial example is given in Figure 4.

For state patterns b having an offset e' in the kernel, we construct the basis of $V(b)$. Then we do a basis transformation resulting in basis vectors in the kernel and outside the kernel. This basis transformation simply consists of taking linear combinations of basis vectors yielding vectors containing one or more orbitals. We can thus partition the basis vectors of V in a set of in-kernel basis vectors generating $V_K = V \cap K$ and a

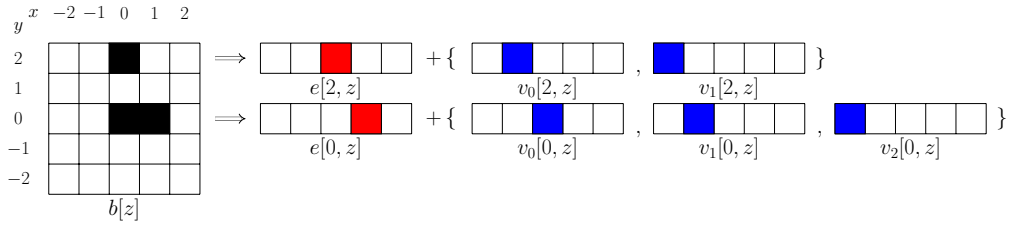


Figure 3: Example of pattern that cannot be forward extended in the kernel. For each active row of slice $b[z]$, offset and basis vectors for the space of χ-compatible rows are depicted in the format "offset + {basis vectors}". Basis for row $y = 2$ has no basis vector with active bit in $x = 1$ to complement the offset at row $y = 0$. Thus, since no offset in the kernel can be constructed, the pattern cannot be extended in the kernel.

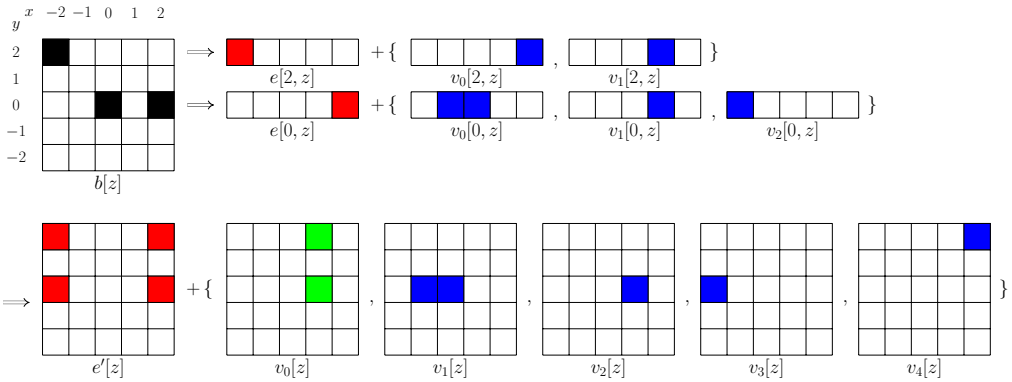


Figure 4: Example of pattern that can be forward extended in the kernel. Above, offset and basis vectors for the space of χ-compatible rows are depicted. Below, offset and basis vectors for the space of χ-compatible slices are depicted, with separation between basis vectors in the kernel (in green) and outside the kernel (in blue). The offset in the kernel can be built as $e'[z] = e[0, z] + e[2, z] + v_0[2, z] + v_2[0, z]$, thus the pattern can be forward extended in the kernel. Vector $v_0[z] = v_1[0, z] + v_2[2, z]$ forms the basis for $V_K[z]$, while $v_1[z], v_2[z], v_3[z], v_4[z]$ form a basis for $V_N[z]$.

set of outside-kernel basis vectors orthogonal generating V_N with $V = V_K + V_N$. This can be done again slice by slice. Specifically, for each z, let $w = \dim(\text{Span}(V[z]))$ and $d = \dim(\text{Span}(V_p[z]))$. We add to $V_N[z]$ d vectors of $V[z]$ whose corresponding parities are linearly independent. Then, for each of the $w - d$ vectors $v \in V[z] \setminus V_N[z]$ we add to it a number of basis vectors of $V_N[z]$ to make it in the kernel. We add the obtained vectors to $V_K[z]$. An example is given in Figure 4.

The procedure described above to generate e in the kernel and a basis for V, with separation between V_K and V_N, is reported in Algorithm 2.

Scanning all elements of $U(b) \cap K$ then simply corresponds to scanning all elements of $e' + V_K$.

In Table 2 we report some experimental results on the extension in the kernel of 2-round trail cores in $|N|$ with weight below 33. We give the number of trails with non-empty intersection with the kernel among all the trails to extend and the dimension of the basis of the affine space and of its intersection with the kernel.

Summarizing, thanks to the fact that most trails cannot be extended in the kernel and that for the remaining ones V_K has low dimension, the cost of forward extension in the kernel is dramatically reduced.

Algorithm 2 Offset and basis creation with separation between in-kernel and outside-kernel vectors

Initialize $\text{flag}_{\text{in\_kernel\_offset}} = \textbf{true}$
for Slices $[z = 0]$ to $[z = w - 1]$ **do**
  Initialize slice offset $e[z] = 0$ and slice basis $V[z] = \emptyset$
  for Rows $[y = 0]$ to $[y = 4]$ **do**
    $e[z] \leftarrow e[z] + e[y, z]$
    $V[z] \leftarrow V[z] \cup V[y, z]$
  end for
  Compute $e_p[z]$ and $V_p[z]$
  if $e_p[z] \in \text{Span}(V_p[z])$ **then**
    $e[z] \leftarrow e[z] + \sum_i v_i$ such that $e[z] \in K$ ▷ where $v_i \in V[z]$
  else
    Set $\text{flag}_{\text{in\_kernel\_offset}} = \textbf{false}$
  end if
  Initialize $V_K[k] = \emptyset$ and $V_N[z] = \emptyset$
  Add to $V_N[z]$ d vectors of $V[z]$ with linearly independent parities
  for All $w - d$ vectors $v \in V[z] \setminus V_N[z]$ **do**
    $v \leftarrow v + \sum_i v_i$ such that $v \in K$ ▷ where $v_i \in V_N[z]$
  end for
end for
$e \leftarrow \sum_z e[z]$, $V_N \leftarrow \bigcup_z V_N[z]$, $V_K \leftarrow \bigcup_z V_K[z]$

Table 2: Results on extension in the kernel. Column two gives the total number of trail cores to extend (both in the forward and backward direction). For both the directions we report the number of trail cores for which V_K is not empty and for such trails the average dimension of V and V_K.

b	# trails to extend	Forward Extension					Backward Extension				
		# trails s.t $V_K \neq \emptyset$	dim V		dim V_K		# trails s.t $V_K \neq \emptyset$	dim V		dim V_K	
			avg	stdev	avg	stdev		avg	stdev	avg	stdev
200	$37 \cdot 10^6$	$9 \cdot 10^3$	14.66	2.95	4.72	1.73	$4 \cdot 10^6$	33.29	5.98	8.37	3.37
400	$30 \cdot 10^6$	$2 \cdot 10^2$	12.19	2.23	4.46	1.43	$5 \cdot 10^5$	33.33	6.07	4.16	2.10
800	$28 \cdot 10^6$	6	9.75	2.24	3.70	0.90	$2 \cdot 10^5$	32.67	6.22	2.61	1.58
1600	$38 \cdot 10^6$	0	0	0	0	0	$2 \cdot 10^5$	32.08	6.25	1.35	1.12

6.1.2 Forward extension outside the kernel

Every element $a \in U$ can be expressed as the sum of the offset e and a subset of the basis vectors, say $A = \{v_i\}$. In particular, $a = e + \sum_{v_i \in A} v_i$. This allows us to arrange the elements of $U(b)$ in a tree in the following way.

First of all, we index the basis vectors of V, establishing an order. Hence a state pattern $a \in U$ is fully determined by the set of indices of the basis vectors in A. If we denote this set by I_a we have $a = e + \sum_{i \in I_a} v_i$. Now, the parent of an element a is obtained by removing the largest element of I_a. In other words, the parent of a state pattern is obtained by *removing its last basis vector*. Or alternatively, the children of a state pattern a are the patterns obtained by adding a basis vector with index larger than the largest included in I_a. The root of the tree is the pattern consisting only of the offset e.

Adding base vectors is not monotonous: adding a basis vector may decrease the weight of $\lambda(a)$. However, the potential weight loss can be bound and this potential weight loss depends on the type of basis vector and its relation with a. Moreover, if the weight of a is high, the worst-case total weight loss (due to the contribution of all base vectors that can

still be added to a) may still give a weight above the limit and the full subtree of a and its descendants can be discarded.

Note that basis vectors outside the kernel either only have a single active bit, or consist of two neighboring active bits in the same row [DV12, Table 3]. The former implies 2 affected columns, the latter 4 affected columns.

As for the worst-case weight loss we can say the following:

- A basis vector in the kernel consisting of n orbitals has a weight loss of at most $4n$. An orbital in a column that is unaffected and that does not overlap with a has no weight loss, an orbital that overlaps with a has a weight loss of at most 2.
- For a basis vector outside the kernel we consider the number of affected columns overlapping with affected columns of a. If this is n, the weight loss is at most $10n+2q$ with q the Hamming weight of the basis vector.

Clearly, in-kernel basis vectors and basis vectors with affected columns not overlapping with those of a have the smallest potential weight loss. We take this into account when deciding the ordering of the basis vectors, putting them last. This leads to a great improvement in forward extension outside the kernel.

6.2 Backward extension

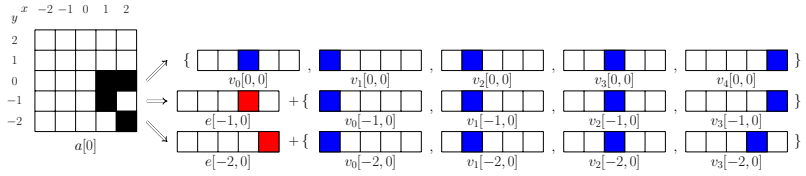
Given a trail core Q starting with state a, backward extension up to some weight T by one round is a scan over all state patterns b such that a is χ-compatible with b and checking whether the weight of the extended trail is below T.

Unlike for forward extension, the set of patterns b that are χ^{-1}-compatible with a do not form an affine space. However, we can define and compute an affine space that contains all such patterns. We call this space the χ^{-1}-*envelope space* of a. As in forward extension, the offset and basis vectors of this space can be generated at row level. For the offset we make use of the property of χ that for an active row in a with a single active bit, the corresponding bit will be active in all states b that are χ^{-1}-compatible with a. Say the active bit in a is in position (x, y, z). Then the offset for this row in b will have a single active bit in position (x, y, z) and there will be four basis vectors each with a single active bit of coordinates (x', y, z), with $x' \neq x$. If the active row in a has more than a single active bit, we consider the 5-dimensional vector space where each basis vector has a single active bit. The resulting χ^{-1}-envelope space of a has an offset with Hamming weight equal to the number of single-bit active rows in a and dimension equal to $5n - \mathrm{HW}(\text{offset})$ with n the number of active rows in a. An example is given in Figure 5a.

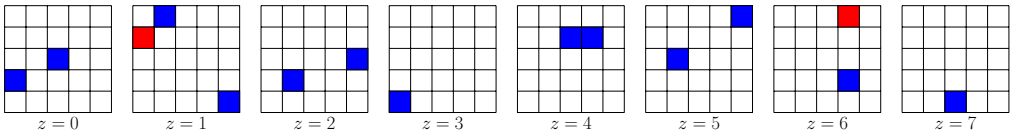
6.2.1 Backward extension in the kernel

As kernel membership is defined at the input and output of θ, we map the χ^{-1}-envelope space of a through π^{-1} and ρ^{-1} to the output of θ. We call the resulting space the *envelope* space of a and denote it as $U = V + e$. Since ρ and π are transpositions, the Hamming weight of the offset e remains the same and the number of active bits in each basis vector remains the same, too. We use techniques similar to those of forward extension to determine whether $U \cap K$ is empty or not. In particular, if e has a slice with odd columns and there are no basis vectors with active bits in that column, then all elements of U will have at least an odd column, disqualifying a for backward extension inside the kernel. An example is given in Figure 5b.

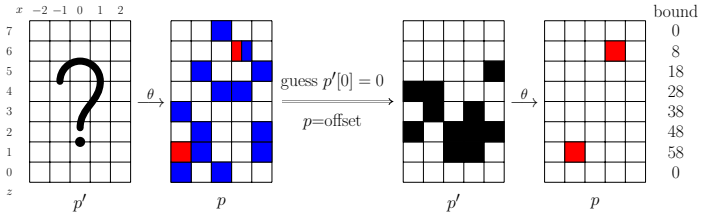
In Table 2 we report the number of trails whose envelope space has non-empty intersection with the kernel. For these trails we report the dimension of the basis of the envelope space and of the intersection with the kernel. As shown, the envelope space is much larger, but the space we concretely investigate is the intersection with the kernel that is significantly smaller, making the search effort feasible.



(a) offsets and basis vectors for χ^{-1}-envelope space



(b) offsets and basis vectors at the output of θ



(c) offset and basis vectors for parity at output of θ

Figure 5: Example on backward extension. In (a) the active rows of a define offset and basis vectors for the χ^{-1}-envelope space. In (b) offset and basis vectors are mapped through $\pi^{-1} \cdot \rho^{-1}$. For the sake of compactness, offset and basis vectors are grouped by slices. This representation allows immediately noticing that there are no basis vectors in slice $z = 1$ with an active bit in column $x = -2$ to complement the offset bit in that column. It follows that a cannot be backward extended in the kernel. In (c) on the left, we derive offset and basis vectors for the space of parity patterns p after θ. In (c) on the right, parity patterns p' and p are incrementally generated and the bound is incrementally computed in the case guess on $p'[0]$ is 0 and p is the offset. The computed value of $p'[1]$ will imply value 11010 for $p'[0]$, which is inconsistent with the guess.

6.2.2 Backward extension outside the kernel

We limit our search of values by restricting the set of parity patterns for $\lambda^{-1}(b)$, that we will denote by a', to investigate. To this end we iteratively construct them, limiting the generation by lower bounding the minimum reverse weight of a'. We start from the envelope space U at the output of θ as in Section 6.2.1. The offset and basis vectors of U define offset and basis vectors for the space of parity patterns at the output of θ. We denote it by $U_p = e_p + V_p$.

Let p be the parity after θ and p' the parity before. Then $p[x, z] = p'[x, z] + p'[x - 1, z] + p'[x + 1, z - 1]$ or, equivalently,

$$p'[x + 1, z - 1] = p[x, z] + p'[x, z] + p'[x - 1, z] \; . \tag{2}$$

Equation 2 allows computing the parity in row $z - 1$ of p' from the parity in row z of p and p'. This can be done recursively. So making an assumption for the value of the parity in some slice of p' allows computing p' completely, for a given p. There are 32 possible assumptions and due to the injectivity of θ, only one of them can be correct.

The knowledge of $p[z]$ and $p'[z]$ allows lower bounding the contribution of slice $a'[z]$ to the minimum reverse weight of a'. Indeed, given the positions of the affected columns

determined by $p[z] + p'[z]$ and the offset and basis of U, we can determine a lower bound for the number of active rows in $a'[z]$ and each active row contributes at least 2 to the minimum reverse weight of a'. We build the parity pattern p' incrementally slice by slice using Equation 2 and keep track of the resulting bound on the minimum reverse weight. For each bit of a basis vector encountered in $p[z]$ we have to consider two possibilities, one with the basis vector present and one with the basis vector absent. So this results again in a tree search where children have a higher cost (lower bound on the minimum reverse weight) than its parent. Hence, we truncate as soon as our partially reconstructed parity exceeds the limit weight. When the scan reaches the value of z where it started and the initial assumption turns out to be correct, it has constructed a valid parity pattern. In that case, we generate all possible states in U that have p as parity and that can be generated by the basis of U. An example on the incremental construction of p and p' (with bound on the minimum reverse weight of a' at each step) is given in Figure 5c and the procedure is translated in Algorithm 3 and Algorithm 4.

Algorithm 3 Generation of valid parity patterns

Initialize $p = e_p$, $p' = 0$
Initialize $L = \emptyset$
for 32 possible row values v **do**
    $p'[0] \leftarrow v$
    Call addRow$(w - 1, p, p', L)$ ▷ see Algorithm 4
end for

Algorithm 4 addRow

if bound on minimum reverse weight is above limit **then**
    return
end if
if $z = w$ **then**
    if (p, p') is consistent **then**
        $L \leftarrow L \cup \{p\}$
    end if
else
    for all $p[z] \in e_p[z] + V_p[z]$ **do**
        $p' \leftarrow p' + p'[z - 1]$ where $p'[z - 1]$ is computed as in Equation 2
        Call addRow$(z - 1, p, p', L)$
    end for
end if

7 Experimental results

We have written new software to implement the techniques described in previous sections and the new code is available as part of the KeccakTools project [BDPV15].

To check correctness of our algorithm, we have tested it against [BDPV15] by generating all 3-round trail cores up to weight 36, i.e. by covering the same space covered in [DV12]. We have found the same number of trails using both codes and thus the same bounds. A comparison between the execution time of our code and the [BDPV15] is given in Appendix D.2.

We have then used our code to cover the space of all 6-round trail cores up to weight $T_6 = 91$, for the widths of KECCAK-f between 200 and 1600. To this end, we covered the

space of all 3-round trail cores up to weight 45.

7.1 2-round trail cores

To generate 2-round trail cores, we apply Lemma 2 and Lemma 3 setting $W = 11$. This implies the generation of 2-round trail cores in the kernel with either w_0 or w_1 smaller than 11 and the generation of all outside-kernel trail cores with $\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 33$. Finally, we generated all 2-round trail cores with either $2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$ or $\mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$ as stated in Lemma 4. Execution times are reported in Appendix A. In Figure 6 we depict the number of 2-round trail cores per weight and per parity profile, to show the trend among the different variants of Keccak-f. The total number of 2-round trail cores generated is reported in Table 7 in the appendix for each parity profile.

7.2 3-round trail cores

For all 2-round trail cores generated, we performed extension in the kernel or outside the kernel, in the backward or forward direction.

In Figure 7 we depict the number of 3-round trail cores per weight and per parity profile, to show the trend among the different variants of Keccak-f. The total number of 3-round trail cores generated is reported in the appendix in Table 8 for each parity profile.

In Figure 8 we then accumulate them and give the total number of 3-round trail cores per weight. Note that each such trail core represents a class of w trail cores that can be obtained by translation along z. In the Appendix (Table 6) we report the execution time for each extension we performed.

Even though Figure 7 only gives a limited view on the number of trail cores up to some weight, it allows us to detect some trends. First, for the number of trails with the smallest weight the pattern appears irregular, but as the weight increases, the number of trails seems to increase exponentially with the weight. Second, we see a dependence on the width. As the width increases, the minimum weight of 3-round trail cores increases, but not in a smooth way. However, the exponent by which the number of trails grows, decreases with the width.

7.3 Extension to 6-rounds

We extended the 3-round trail cores found to 6 rounds up to limit weight $2T_3 + 1 = 91$. We found no 6-round trail cores with weight below or equal to the limit weight. This means that 92 is a lower bound for the weight of 6-round differential trails in Keccak-f for widths higher than 100. For Keccak-f[1600] this results is an improvement of the bound found in [DV12], which was 74. For the other variants it is a new result.

Scanning the space of 3-round trail cores up to weight 45, gives bounds for 4 and 5-round trail cores up to weight 47 and 49 respectively. For Keccak-f[200] there exists a 4-round trail of weight 46, which is thus the lightest trail. For Keccak-f above 200 no trail is found below the limit, therefore we can say that the lower bound on the weight of 4-round trail cores for these variants is 48. As for 5-rounds, for all Keccak-f from 200 to 1600 no trail core with weight smaller or equal to 49 is found, therefore we can say that a lower bound on the weight of 5-round trail cores is 50.

During our search we have found some interesting trail cores exceeding the limit weight, but that are, to the best of our knowledge, the lightest trails found up to now with no particular symmetry properties. We report them in Table 3 giving their weight and parity profiles. These trails allow to define upper bounds for the minimum weight of trails. In the definition of these upper bounds, we must take into account the Matryoshka structure [BDPV11]. For instance, for the 6-round trail of weight 278 in Keccak-f[400] there will be a corresponding trail in Keccak-f[800] of weight $2 \cdot 278$ and in Keccak-f[1600] of

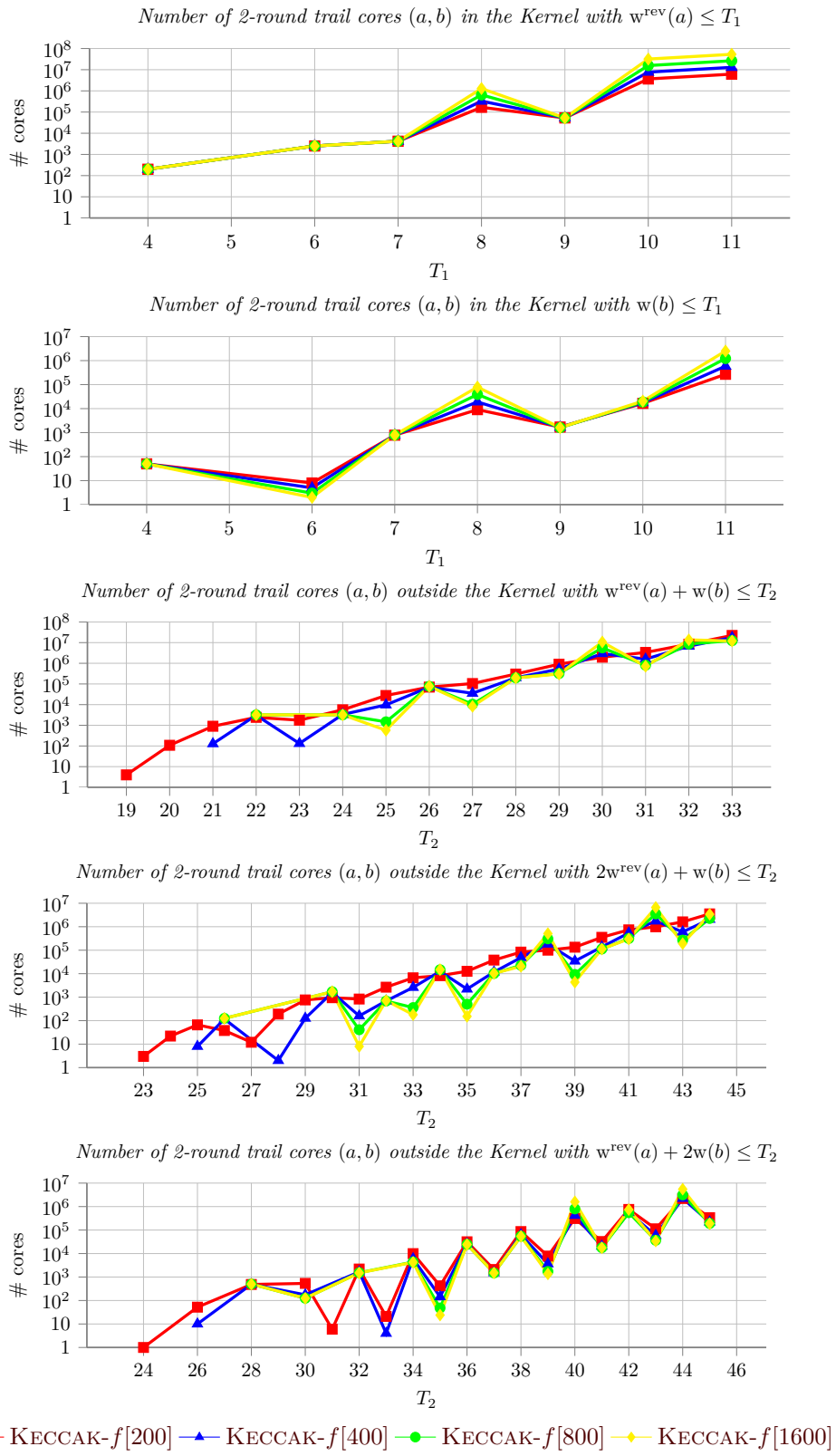


Figure 6: Number of 2-round trail cores for different cost functions.

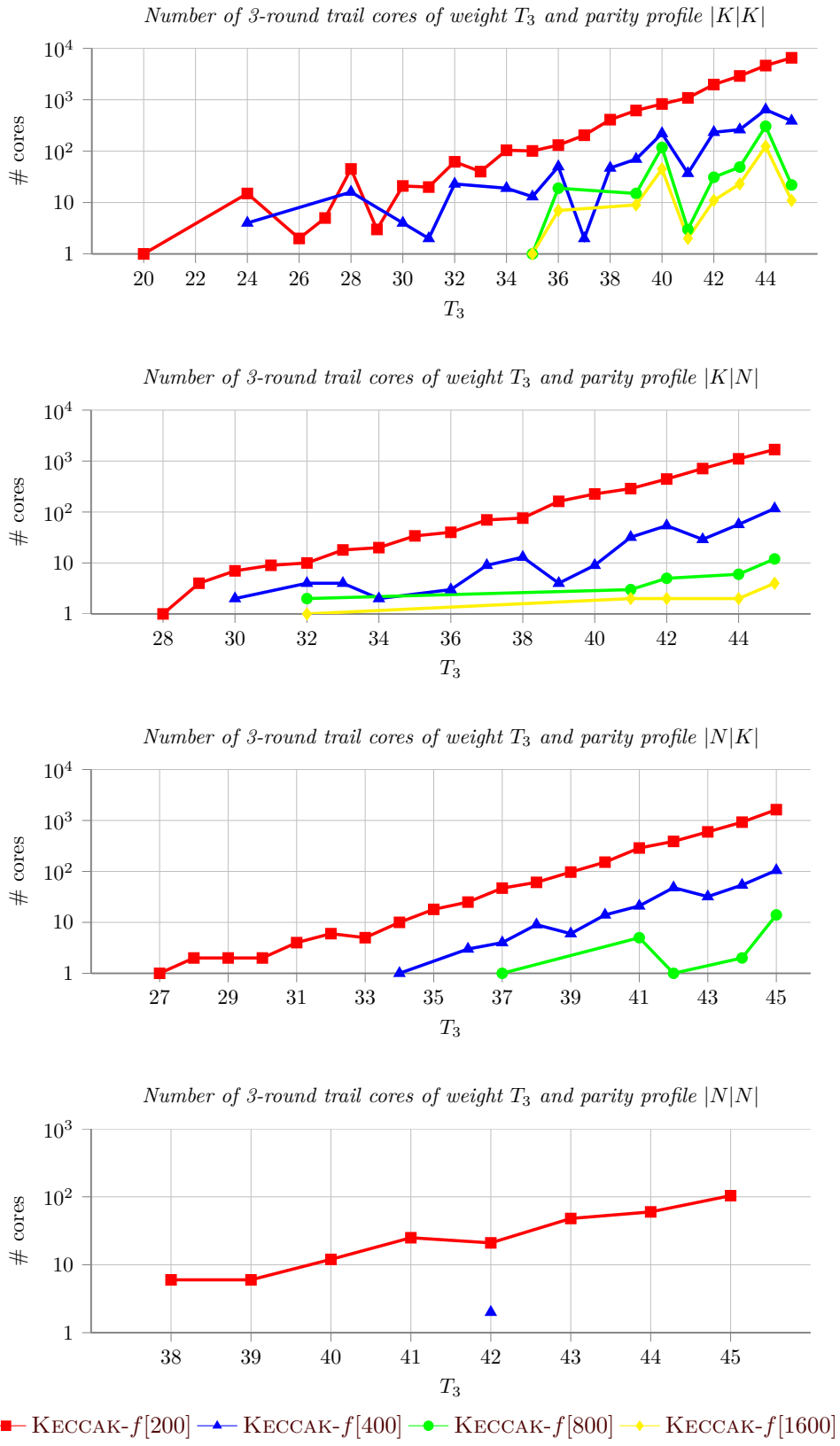


Figure 7: Number of 3-round trail cores with weight T_3 for different parity profiles.

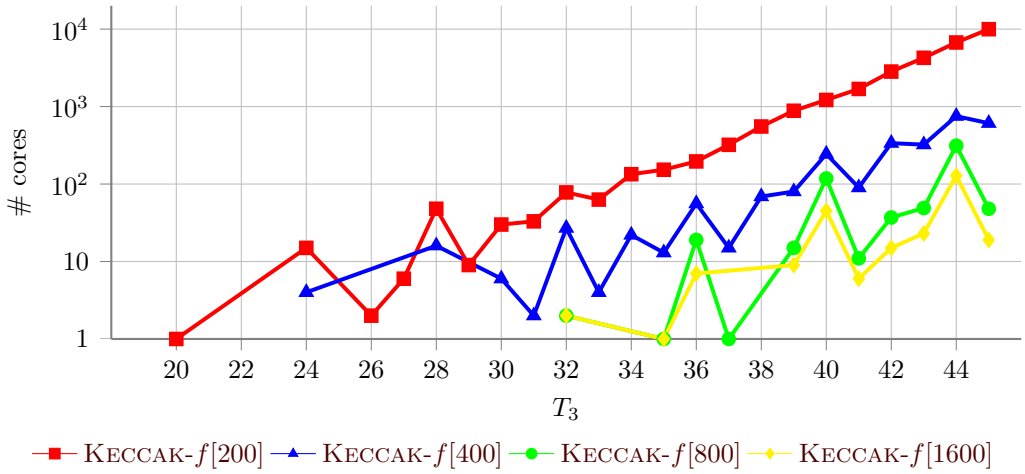


Figure 8: Number of all 3-round trail cores with weight T_3.

Table 3: Lightest trails found for different variants of Keccak-f.

rounds	profiles	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
3	weight	4 8 8	8 8 8	4 4 24	4 4 24
	parity	$\|K\|K\|$	$\|K\|K\|$	$\|K\|N\|$	$\|K\|N\|$
4	weight	21 9 8 8	29 6 4 24	20 9 8 67	16 13 12 93
	parity	$\|N\|K\|K\|$	$\|N\|K\|N\|$	$\|N\|K\|N\|$	$\|K\|K\|N\|$
5	weight	36 13 9 15 16	67 14 12 16 38	124 21 14 11 77	166 22 16 16 152
	parity	$\|N\|N\|K\|K\|$	$\|N\|K\|K\|N\|$	$\|N\|K\|K\|N\|$	$\|N\|K\|K\|N\|$
6	weight	45 44 13 9 15 16	81 35 6 4 24 128	192 17 12 12 73 438	166 22 16 16 152 841
	parity	$\|N\|N\|N\|K\|K\|$	$\|N\|N\|K\|N\|N\|$	$\|N\|K\|K\|N\|N\|$	$\|N\|K\|K\|N\|N\|$

weight $2 \cdot 556$. In this case, the upper bound is given by this trail and not by the trail reported in Table 3.

We recap the obtained results in Table 1 reporting the minimum weight of trails or the range where the minimum weight lives, for the variants of Keccak-f from 200 to 800.

7.4 Bounds on n_r-rounds

From the bounds we have found, we can derive lower bounds on the total number of rounds for each variant of Keccak. We report them in Table 4. Each bound is derived considering the worst case scenario when extending 6-round trails to n_r rounds. For instance, for Keccak-f[400], the total number of round is 20. A 20-round trail can be obtained by first combining three 6-round trails that in the worst case contribute only $3 \cdot 92 = 276$. Then, for the two rounds left, the worst case is when they are one a the beginning and one at the end of the trail, thus contributing only 2 to the weight. This gives a lower bound of 280.

The obtained results confirm that the security of Keccak relies not on the difficulty of finding exploitable trails, but rather on their absence.

From the lower bound on 6 rounds we can also infer a lower bound of 184 for 12 rounds, which is the number of iterations used in Keyak.

8 Conclusions

We presented new techniques to scan the whole space of 3-round differential trails in Keccak-f up to a given weight. At first, we introduced a general formalism that allows to generate differential patterns as a simple and efficient tree traversal. This technique

Table 4: Bounds on the weight of trails over the total number of rounds.

	b=200	b=400	b=800	b=1600
n_r	18	20	22	24
bound	276	280	292	368

is of independent interest and could be applied to those primitives with a linear and a non-linear mixing layers.

For the specific case of KECCAK-f, we instantiated the tree traversal method exploiting the propagation properties of the step mappings and their translation-invariance along the z-axis. Finally, we presented new techniques that exploit the parity profile of trails to extend them efficiently.

We implemented these methods and used them to find lower bounds for differential trail weights for KECCAK-$f[200]$ to KECCAK-$f[1600]$. The result obtained for KECCAK-$f[1600]$ over 6 rounds improves known results, whereas the results for KECCAK-$f[b]$ for $b \in \{200, 400, 800\}$ are new.

Furthermore, the results provide insight in how the number of exploitable 3-round trails scales with the width of the permutation. It appears that low-weight trails over more than 4 rounds become sparser with increasing width. This makes sense as the sheer number of 3-round trails to be extended decreases with the width and the probability that a given 3-round trail can be extended with a low-weight pattern decreases with the width.

As rounds are combined, the number of trails up to a given weight per round decreases significantly. As shown in Figure 1, for KECCAK-$f[1600]$ the number of single-round differentials with weight up to 15 is $2^{60.7}$. The number of 2-round trails with weight up to 30 is $2^{23.6}$, whereas the number of 3-round trails up to weight 45 is just 2^8. After extending to 6 rounds, we see that there are no such trails.

References

[BDP+14] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. CAESAR submission: KETJE v1, March 2014. http://ketje.noekeon.org/.

[BDP+15] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. CAESAR submission: KEYAK v2, August 2015. http://keyak.noekeon.org/.

[BDPV11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The KECCAK reference, January 2011. http://keccak.noekeon.org/.

[BDPV15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. KECCAKTOOLS software, October 2015. http://keccak.noekeon.org/.

[BFL10] C. Bouillaguet, P.-A. Fouque, and G. Leurent. Security analysis of SIMD. In A. Biryukov, G. Gong, and D. R. Stinson, editors, *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, volume 6544 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2010.

[BS90] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In A. Menezes and S. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[DPVR00] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: the block cipher NOEKEON. Nessie submission, 2000. http://gro.noekeon.org/.

[DR01] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In Bahram Honary, editor, *Cryptography and Coding, 8th IMA International Conference, Cirencester, UK, December 17-19, 2001, Proceedings*, volume 2260 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2001.

[DR02] J. Daemen and V. Rijmen. *The design of Rijndael — AES, The Advanced Encryption Standard*. Springer-Verlag, 2002.

[DV12] J. Daemen and G. Van Assche. Differential propagation analysis of Keccak. In A. Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 422–441. Springer, 2012.

[MP13] N. Mouha and B. Preneel. Towards finding optimal differential characteristics for ARX: Application to Salsa20. *IACR Cryptology ePrint Archive*, 2013:328, 2013.

[MWGP11] N. Mouha, Q. Wang, D. Gu, and B. Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In C. Wu, M. Yung, and D. Lin, editors, *Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers*, volume 7537 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.

[SHW+14] S. Sun, L. Hu, M. Wang, P. Wang, K. Qiao, X. Ma, D. Shi, and L. Song. Towards finding the best characteristics of some bit-oriented block ciphers and automatic enumeration of (related-key) differential and linear characteristics with predefined properties. *IACR Cryptology ePrint Archive*, 2014:747, 2014.

A Execution times

In Table 5 we report the execution time needed to generate the different sets of 2-round trail cores, while in Table 6 we report the time required to extend those trails in the backward or forward direction inside or outside the kernel. Experimental results have been obtained running the code on an Intel Xeon X5660 processor running at 2.8GHz, using a single core for each experiment.

Table 5: Execution time for the generation of 2-round trail cores. Last line refers to the technique of [DV12].

2-round trail cores	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\lvert K\rvert\, \mathrm{w}^{\mathrm{rev}}(a) \leq 11$	2m20s	6m37s	19m50s	6h06m10s
$\lvert K\rvert\, \mathrm{w}(b) \leq 11$	5s	14s	44s	2m27s
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 33$	1h36m51s	1h22m45s	1h42m27s	2h58m43s
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	4h09m18s	2h50m14s	3h46m41s	6h44m40s
$\lvert N\rvert\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	12h19m54s	25m36s	37m14s	1h12m48s
$\lvert K\rvert$ with $\chi(b)$ in $\lvert K\rvert$	37h26m17s	12h38m33s	11h29m54s	16h11m16s
Total	55h34m45s	17h23m59s	17h56m06s	33h16m04s

Table 6: Execution time for the extension of 2-round trail cores to 3 rounds. Symbols $\rightarrow |K|$ and $\rightarrow |N|$ denote forward extension in the kernel and outside the kernel respectively; $\leftarrow |K|$ and $\leftarrow |N|$ denote backward extension in the kernel and outside the kernel respectively.

2-round trail cores	extension	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\lvert K\rvert\, \mathrm{w}^{\mathrm{rev}}(a) \leq 11$	$\rightarrow \lvert N\rvert$	18h03m01s	1h40m36s	19h11m45s	136h01m47s
$\lvert K\rvert\, \mathrm{w}(b) \leq 11$	$\leftarrow \lvert N\rvert$	17m57s	5m55s	31m13s	2h36m01s
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 33$	$\rightarrow \lvert K\rvert$	2m51s	2m25s	3m17s	6m32s
	$\leftarrow \lvert K\rvert$	18h17m07s	2h16m52s	17h40m33s	33h28m14s
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	$\leftarrow \lvert N\rvert$	8h54m39s	39m34s	4h19m50s	45h52m52s
$\lvert N\rvert\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	$\rightarrow \lvert N\rvert$	1h00m39s	24m37s	2h14m51s	10h06m26s
$\lvert K\rvert$ with $\chi(b)$ in $\lvert K\rvert$	$\rightarrow \lvert K\rvert$	3s	1s	2s	5s
Total		46h36m17s	5h10m00s	43h30m18s	225h35m56s

B Number of 2 and 3-round trail cores generated

We report here the total number of trails generated per parity profile. The same information can be extracted from Figures 6, 7 and 8.

Table 7: Number of 2-round trail cores generated for each parity profile.

2-round trail cores	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\lvert K\rvert\, \mathrm{w}^{\mathrm{rev}}(a) \leq 11$	$1.0 \cdot 10^7$	$2.1 \cdot 10^7$	$4.3 \cdot 10^7$	$8.6 \cdot 10^7$
$\lvert K\rvert\, \mathrm{w}(b) \leq 11$	$3.0 \cdot 10^5$	$6.3 \cdot 10^5$	$1.3 \cdot 10^6$	$2.6 \cdot 10^6$
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 33$	$3.7 \cdot 10^7$	$3.0 \cdot 10^7$	$2.8 \cdot 10^7$	$3.8 \cdot 10^7$
$\lvert N\rvert\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 45$	$3.9 \cdot 10^6$	$3.6 \cdot 10^6$	$4.8 \cdot 10^6$	$8.2 \cdot 10^6$
$\lvert N\rvert\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 44$	$7.6 \cdot 10^6$	$5.4 \cdot 10^6$	$6.8 \cdot 10^6$	$1.1 \cdot 10^7$
$\lvert K\rvert$ with $\chi(b)$ in $\lvert K\rvert$	$3.6 \cdot 10^4$	$3.1 \cdot 10^3$	$5.2 \cdot 10^2$	$1.3 \cdot 10^2$
Total	$5.9 \cdot 10^7$	$6.0 \cdot 10^7$	$8.4 \cdot 10^7$	$1.5 \cdot 10^8$

Table 8: Number of 3-round trail cores up to weight 45.

3-round trail cores	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
$\|K\|K\|$	$2.0 \cdot 10^4$	$2.0 \cdot 10^3$	$5.6 \cdot 10^2$	233
$\|K\|N\|$	$4.9 \cdot 10^3$	$3.4 \cdot 10^2$	28	11
$\|N\|K\|$	$4.3 \cdot 10^3$	$3.0 \cdot 10^2$	23	0
$\|N\|N\|$	$2.8 \cdot 10^2$	2	0	0
Total	$2.9 \cdot 10^4$	$2.7 \cdot 10^3$	$6.1 \cdot 10^2$	244

C Proof of Lemma Lemma 7

Proof. First of all, it cannot overestimate the cost of a parity-bare state. This follows from the fact that $\bar{a}$ and $\bar{b}$ are obtained by removing bits at a and/or at b, so the weight of reduced states is smaller than the weight of original states, since the weight functions $\mathrm{w}^{\mathrm{rev}}(a)$ and $\mathrm{w}(b)$ are monotonic in the addition of bits [BDPV11]. It follows that $L(a) \leq \alpha \cdot \mathrm{w}^{\mathrm{rev}}(\bar{a}) + \beta \cdot \mathrm{w}(\bar{b}) \leq \alpha \cdot \mathrm{w}^{\mathrm{rev}}(a) + \beta \cdot \mathrm{w}(b)$.

It remains to prove, that the cost function lower bounds the weight of descendants.

For those descendants obtained by adding column assignments that do not remove bits from a and b, the weight is lower bounded by the weight of the reduced states, since they have less active bits. For descendants obtained by adding an odd-0 column to an affected even column, the active bit in $y = 0$ at a can be moved to b or vice versa. The reduced states don't have these bits, so again the weight of reduced states lower bounds the weight of such descendants.

The only cases where bits can be removed from reduce states are when affected even columns are added to odd-0 columns.

For those columns not in N_0: if the bit in $y = 0$ of the affected even column is passive, then the bit remains active at a and become passive at b. The weight of such states is lower bounded by the weight of reduced states because they don't have such bits at b. If the bit in $y = 0$ of the affected even column is active, then the bit at b remains active and the bit at a is moved to another row, not changing its contribution to the weight.

For those columns in N_0: if the bit in $y = 0$ of the affected even column is passive, then the bit remains active at a and become passive at b. This may reduce the weight by a factor 2β. If the bit in $y = 0$ of the affected even column is active, then the bit at b remains active and the bit at a is moved to another row, possibly reducing the weight by at most a factor of 2α. The maximum weight reduction is thus by a factor of $2 \max(\alpha, \beta)$. This can happen at most $\#N_0$ times and hence the weight can never become smaller than the weight of the reduced states minus $2 \cdot \max(\alpha, \beta) \cdot \#N_0$. □

D Comparison with previous work

One may wonder why we didn't just use the techniques described in [DV12] and implemented in the [BDPV15] to cover the space of 3-round trails up to weight 45. This is actually what we tried to do as our first step. But we encountered a number of obstacles, which motivated us to develop new and more efficient techniques. We describe them here and report the execution time needed to cover the space of 3-round trails up to weight 36 in Keccak-f[1600] using both approaches. These results will show how the new techniques presented in this work allow to scan the space of trails more efficiently, thus allowing to push the target weight to higher values.

Table 9: Ratio between the number of parity patterns that give rise to trails with weight below T_2 and the number of parity patterns with bound below T_2, according to [DV12].

T_2	$b = 200$	$b = 400$	$b = 800$	$b = 1600$
26	22/181	18/61	15/37	15/35
28	30/534	27/127	24/48	22/41
30	42/1505	34/364	31/99	31/87
32	66/4028	45/938	36/342	35/277

D.1 Limitations of the previous techniques

In [DV12], the space of 3-round trail cores (a_1, b_1, a_2, b_2) up to weight 36 is covered by splitting it into subspaces that are scanned using different techniques:

1. all 3-round trail cores with both a_1 and a_2 in the kernel. This is covered by first generating all 2-round trail cores with a in the kernel for which there exists a compatible state $\chi(b)$ in the kernel. Then, forward extension (in the kernel) of these trails is performed.

2. all 3-round trail cores such that $\mathrm{w}^{\mathrm{rev}}(a_1) < 8$ or $\mathrm{w}(b_1) < 8$ or $\mathrm{w}(b_2) < 8$. This set is covered by first generating all 2-round trail cores such that a or b has three or less active rows. In fact, a weight up to 7 implies at most 3 active rows, since each row contributes at least 2 to the weight. Then, forward and backward extension of these trails is performed.

3. all 2-round trail cores such that $\mathrm{w}^{\mathrm{rev}}(a_1) \geq 8$, $\mathrm{w}(b_1) \geq 8$ and $\mathrm{w}(b_2) \geq 8$ and not both a_1 and a_2 in the kernel. This is covered by first generating all 2-round trail cores outside the kernel with weight up to 28 and then extending them in forward and backward direction.

While trying to use these methods with a limit weight above 36, we have faced the following bottlenecks:

1. in point 2 above: going for weight 8 or more explodes the number of cases to investigate and to extend. In fact, in Keccak-f[1600] the number of states with up to 3 active rows is about 2^{30}. Considering 8 or more implies to generate all states with up to 4 or more active rows, and thus to build more than 2^{40} states.

2. in point 3 above: the generation of 2-round trail cores outside the kernel is performed in two steps. In the first step parity patterns are incrementally generated by adding runs and computing a lower bound on the weight of trails with given parities. Only those parity patterns with small lower bound (namely, below 28) are kept. Namely, only those parity patterns that might give rise to a trail core with weight smaller than 28. In the second step, state patterns are concretely built for those parities generated during the first step. Lemma 3 in [DV12] is tailored to the initial target weight of 36; aiming for a bigger target gives rise to thousands of parity patterns to investigate, most of which do not actually give rise to valid 2-round trail cores. In fact, the lower bound computed in the first step is loose and the result is that many of the parity patterns generated in the first step do not actually give rise to trails whose weight is below the limit. In Table 9 we report the ratio between the number of parity patterns that really give rise to trails with weight below T_2 and the number of parity patterns with bound below T_2, for different values of T_2.

   The gap between the lower bound and the actual minimum weight of trails per given parity, suggests the need for better techniques to lower bound the weight. We decided

to incrementally generate state patterns making use of the run structure of parity patterns. This allows us to get better bounds on the weight of resulting trails, since the new bound is computed from the actual weight of the current trail instead of being computed only from lower bounds on runs as in [DV12].

3. in general, increasing the target weight results in the generation of a bigger amount of 2-round trails which must be extended to 3 rounds. Moreover, the bigger the weight of a state is, the bigger the number of (χ or χ^{-1}) compatible states is. Therefore, increasing the target weight makes extension much more expensive because the number of trails to investigate and the number of compatible states for each trail grows. This motivated us to optimize extension, in order to avoid the investigation of patterns that will not give rise to valid trails (i.e. below the target weight).

D.2 Covering the space of 3-round trails up to weight 36

In order to compare our new techniques to previous work, we repeated the experiments described in [DV12]. In particular, we generated all 3-round trail cores for KECCAK-f[1600] up to weight 36 using the routines available in [BDPV15]. Then, we covered the same space using our code, by setting $T_1 = 8$, $T_2 = 27$ and $T_3 = 36$.

In Table 10 and Table 11 we reported the execution time for both the approaches.

Table 10: Execution time for the generation of 3-round trail cores up to weight 36 using the techniques of [DV12] and the KECCAKTOOLS.

generation of 2-round trail cores		extension to 3 rounds	
type	time	type	time
$\|a\|_{row} \leq 3$	34h38m14s	$\rightarrow$	2h07m13s
		$\leftarrow$	32m28s
$\|b\|_{row} \leq 3$	14h55m43s	$\rightarrow$	7s
		$\leftarrow$	more than 10 days
$\|N\|\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 28$	3m28s	$\rightarrow$	2h00m50s
		$\leftarrow$	more than 10 days
$\|K\|$ with $\chi(b)$ in $\|K\|$	4m50s	$\rightarrow$	4s
Total	49h42m15s	Total	more than 20 days

Table 11: Execution time for the generation of 3-round trail cores up to weight 36 using the new techniques presented in this work. Last line refers to the technique of [DV12].

generation of 2-round trail cores		extension to 3 rounds	
type	time	type	time
$\|K\|\, \mathrm{w}^{\mathrm{rev}}(a) \leq 8$	53s	$\rightarrow \|N\|$	39m53s
$\|K\|\, \mathrm{w}(b) \leq 8$	9s	$\leftarrow \|N\|$	3m56s
$\|N\|\, \mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 27$	1m23s	$\rightarrow \|K\|$	2m36s
		$\leftarrow \|K\|$	5m06s
$\|N\|\, \mathrm{w}^{\mathrm{rev}}(a) + 2\mathrm{w}(b) \leq 36$	4m27s	$\leftarrow \|N\|$	2m56s
$\|N\|\, 2\mathrm{w}^{\mathrm{rev}}(a) + \mathrm{w}(b) \leq 35$	32s	$\rightarrow \|N\|$	48s
$\|K\|$ with $\chi(b)$ in $\|K\|$	4m50s	$\rightarrow \|K\|$	4s
Total	12m14s	Total	55m19s