When the Wrong Key Lives On: The Key-Recovery Procedure in Integral Attacks

Christof Beierle, Gregor Leander, and Yevhen Perehuda

Faculty of Computer Science, Ruhr University Bochum, Bochum, Germany
firstname.lastname@rub.de

Abstract. An integral distinguisher for a block cipher is defined by a nontrivial subset of plaintexts for which the bitwise sum of (parts of) a certain internal state is independent of the secret key. Such a distinguishing property can be turned into a key-recovery procedure by partially decrypting the ciphertexts under all possible keys and then filtering the key candidates using the integral distinguisher. The behavior of this filter has never been analyzed in depth, and we show that the ubiquitous hypothesis about its behavior is incorrect.
Fortunately, the deviation is either limited or can be lifted to improve the underlying attacks. By algorithmically determining the exact subspaces of key candidates to be guessed – whose dimensions are often lower than expected – we are able to improve upon the best known integral key-recovery attacks on various ciphers.

Keywords: Block cipher · Integral attack · Key Recovery · Wrong-Key Randomization · Linear Structure

1 Introduction

The continuous development of new attacks and the refinement of our understanding of existing attack vectors is essential for maintaining trust in cryptographic primitives. This approach has been highly successful, particularly in the case of symmetric cryptography, where we are now in the fortunate position of having ciphers and design frameworks that resist all known attack vectors while offering excellent performance. Nevertheless, even many classical attacks – known and studied for several decades – still warrant our attention, as they continue to yield interesting, and sometimes surprising, new insights.

This is especially true for classical attacks beyond the most extensively studied linear and differential techniques. In this work, we focus on the most prominent among them: *integral attacks*.

Integral attacks, which grew out of the study of *high-order differentials*, were first proposed independently by Lai [23] and Knudsen [20], and later systematized by Knudsen and Wagner with the "SQUARE" attack [21]. Initially, an integral distinguisher for a family of block ciphers is obtained by identifying a nontrivial subset of $\mathbb{F}_2^n$ of plaintexts for which the bitwise sum of (parts of) the

©IACR 2026. This is the full version of the article submitted by the authors to the IACR and to Springer-Verlag on February 21, 2026. The version published by Springer-Verlag is available at https://doi.org/10.1007/978-3-032-25333-0_2.

corresponding ciphertexts (or of some internal state, when key-recovery rounds are appended) is independent of the secret key k. Such a property immediately yields a distinguisher and, with additional analysis, may be turned into a key-recovery procedure. For this, one partially decrypts the ciphertexts under all possible key candidates and filters the key candidates using the integral distinguisher. The number of keys to be guessed and the probability of filtering wrong key candidates are the main parameters for the attack.

There has been substantial progress since these early works. In particular, the *division property* (first proposed in [28]) provides a powerful and systematic framework for detecting integral-like properties; follow-up work both extended and automated many of these techniques. *Monomial prediction* is one of the main advances here and has been proven useful in practice (see, e.g., [18]). More recently a geometric viewpoint has been advanced by Beyne et al. [7], which not only captures bitwise XOR, i.e., divisibility by two, but can be extended to divisibility by 2^v through a careful 2-adic analysis.

Complementing constructive techniques, considerable effort has gone into arguments for guaranteeing security by stating conditions under which integral distinguishers cannot exist. Such *integral resistance* arguments are central to modern security assessments. For key-alternating ciphers, Hebborn et al. [17] gave the security arguments against integral distinguishers under the assumption of independent round keys; these arguments were later generalized to cover word-wise modular key addition in [5].

A notable gap in the literature is the systematic study of the *key-recovery* step associated with integral distinguishers. While many works carefully derive distinguishers and bounds on data/time complexity, the behaviour of wrong keys in an integral-based key search has received much less attention than in, say, differential or linear cryptanalysis. The understanding of how internal-state sums behave under incorrect key guesses – and how rapidly such behaviour diverges from the correct-key case – is important both for practical attacks and for principled security proofs, but it is not well studied in general.

Key-Recovery and the Wrong-Key-Randomization Hypothesis. The most common way to lift a distinguisher into a key-recovery attack is to append (or prepend, or both) a small number of rounds to the distinguisher. Given a set of ciphertexts, one guesses the relevant key bits, computes backwards through the appended rounds to the distinguisher's output, and checks whether the distinguishing property holds. Keys for which the property holds are retained as candidate keys; all others are discarded. Again, there has been substantial work on improving the algorithmic part of computing backwards, e.g., by using a FFT-like approach [29], partial sums [12], or their combination [11].

Depending on the distinguisher, this procedure yields a (hopefully) small set of candidates – the expected size of this set captures the attack's gain – and (hopefully) contains the correct subkey – the probability of this happening is captured in the attack's success probability. The expected size and the success probability are the two main metrics that determine an attack's efficiency, so estimating them as precisely as possible is of great interest. Computing the

success probability is often straightforward; in the case of integral attacks, it is trivial, since the distinguishing property always holds for the correct key. In other words, for integral attacks, the correct key is guaranteed to be among the surviving candidates.

The behaviour under wrong-key guesses is much more delicate, and is often glossed over. A common simplifying assumption is that a wrong key produces internal states that are uniformly random from the distinguisher's perspective. Intuitively, this corresponds to treating the effect of a wrong key as adding further independent rounds of the cipher, which yields states indistinguishable from random. This assumption makes it easy to build simple statistical models for the expected number of remaining candidates. However, this uniformity assumption is rarely justified rigorously, and as we will see often wrong, especially if the number of appended rounds is small. For differential and linear attacks, considerable work has been devoted to validating and refining such assumptions [1,9,27,10].

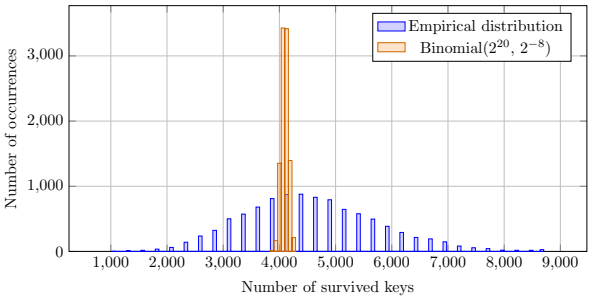
For integral cryptanalysis, however, to the best of our knowledge and surprise, this has not been addressed at all. As a matter of fact, many previously discussed attacks deviate significantly from the expected behavior, as we showcase in the following with a very recent attack on `PRESENT`.

An Illustrative Example. An interesting example is the key-recovery attack (Fig. 6) from Beyne et al. [7, Sec. 7.4] that shows discrepancy between theoretical and factual key recovery. Using a six-round integral distinguisher with data complexity 2^{12} on the least significant nibble, the paper derives a key-recovery filter of 2^{-8} while guessing 20 key bits. Thus, under the usual assumption, the number of "surviving" key guesses should follow

$$\text{Binomial}\big(N = 2^{20},\, p = 2^{-8}\big),$$

with mean $Np = 2^{12} = 4096$ and standard deviation $\sqrt{Np(1-p)} \approx 63.87$.

However, in 10,000 independent experiments, we observed that the survivor count's variance is far larger (empirical std ≈ 1213 vs. the predicted ≈ 64), as depicted below.



This is mainly caused by a strong clustering: each key is indistinguishable from 255 additional keys. As we will discuss in detail later, this clustering is due to key information (not necessarily single key bits or information restricted to a single round key) that is irrelevant to the outcome of the distinguisher.

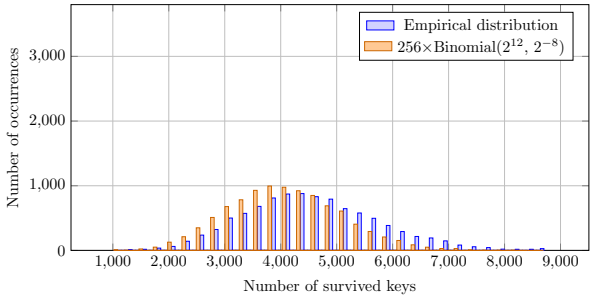
Accounting for this clustering yields a corrected model: The 2^{20} candidates partition into $2^{20}/256 = 2^{12}$ equivalence classes; each class survives independently with probability $p = 2^{-8}$, contributing a block of size 256 when it does. Hence, the adjusted survivor distribution is

$$256 \times \text{Binomial}\big(N' = 2^{12},\, p = 2^{-8}\big).$$

This preserves the mean ($256 \cdot N'p = 4096$) but multiplies the variance by 256, so the predicted standard deviation becomes

$$\sqrt{256^2\, N'p(1-p)} \;=\; \sqrt{1{,}044{,}480} \;\approx\; 1022,$$

which matches the experiments closer, but still leaves a visible shift in the empirical mean (≈ 4521 vs. the theoretical 4096) as shown below.



This shift in the expectation clearly requires further investigation.

Our Contribution

For the key-recovery procedure in integral attacks, we study, explain, and exploit the deviation from the expected behavior. We consider the generalized notion of integral distinguishers from [7], which asserts that the (integer) sum of a certain internal state bit is divisible by some fixed power of two.

After fixing our notation and recalling the most important concepts in Section 2, we first focus on the shift in the expectation. In Section 3, we consider the case where the partial decryption actually provides a random permutation. We reduce this problem to a non-trivial but fairly classical combinatorial computation. Interestingly, even in this case, the filtering problem does not coincide with what has been used in the literature so far. While we observe strong deviations in some cases, it can fortunately be proven that the deviation from the expectation remains small in the most important cases. We then elaborate on the question of a non-ideal partial decryption. Here, we provide an interesting link to the *Differential-Linear Connectivity Table* (DLCT) [3], which explains the experimental deviations.

We then turn our attention to the clustering of keys. In a nutshell, two key guesses k and $k+w$ behave in the same way from the distinguisher's perspective if w is a *linear structure* in the key of the partial decryption function. While this

phenomenon has been studied and exploited before, it has by no means been used to its full potential. We extend this line of work by introducing the notion of *affine related-key subspace trails (ARKST)* in Section 4. We develop an efficient algorithm that automatically computes all such trails for an SPN with independent round keys for up to at least three (key-recovery) rounds. This allows us to algorithmically find the linear structures in the key and therefore to precisely capture the key information that does not influence the partial decryption used in the distinguisher.

Applying this framework, we improved upon the best-known integral key-recovery attacks on `PRESENT`, `GIFT`, and `RECTANGLE` (Section 5), and – by analyzing the algebraic normal form – on `SIMON32` (analysis given in Supplementary Material A). In all these cases, we rely on the known distinguishers. The resulting improvements are summarized in Table 1. While the improvements for `GIFT` and `SIMON32` are rather limited, we obtain substantial gains on the attacks against `PRESENT` and `RECTANGLE`. The main reason is that in these cases there exists key information that does not influence the computation, yet spreads over several rounds; something that has not been captured before.

Table 1: Comparison of the best integral key-recovery attacks.

Cipher	Rounds	Data	Time	Technique and Source
`PRESENT`	12	2^{60}	$2^{48.9}$ Add.	MTTS & FFT [30]
	12	2^{60}	$\mathbf{2^{28.3}}$ Add.	Section 5.1
	12	2^{63}	$2^{93.6}$ Add.	FFT based on [7]
	12	2^{63}	$\mathbf{2^{53}}$ Add.	Section 5.1
`GIFT-64`	14	2^{63}	2^{96} Enc.	Partial sums [2]
	14	2^{63}	$\mathbf{2^{94}}$ Enc.	Section 5.2
	15	2^{63}	2^{96} Enc.	Partial sums based on [33]
	15	2^{63}	$\mathbf{2^{94}}$ Enc.	Section 5.2
`RECTANGLE`	13	2^{63}	$2^{62.75}$ Add.	FFT based on [24,33]
	13	2^{63}	$\mathbf{2^{54.6}}$ Add.	Section 5.3
	14	2^{63}	$2^{127.6}$ Add.	FFT based on [24,33]
	14	2^{63}	$\mathbf{2^{115.5}}$ Add.	Section 5.3
`SIMON32`	21	2^{31}	$2^{54.59}$ Enc.	MITM & Partial sums [31]
	21	2^{31}	$\mathbf{2^{52.75}}$ Enc.	Supplementary Material A

2 Preliminaries

When attacking an iterative block cipher, a typical approach is to consider the cipher under investigation as decomposed into a cascade of two parts,

$$E_k \circ E'_{k'},$$

together with a distinguishing property $\mathcal{D}$ that holds after the application of $E'_{k'}$. Typically, this property $\mathcal{D}$ is assumed to be satisfied independently of the choice of the subkey k'. The adversary is assumed to have access to ciphertexts

$$c = E_k \circ E'_{k'}(m)$$

for either known or chosen plaintexts m, depending on the attack model. The objective is to recover the subkey k that produced the observed ciphertexts.

The high-level attack strategy proceeds by guessing candidate keys k_{cand} for k, partially decrypting the ciphertexts as $E^{-1}_{k_{\text{cand}}}(c)$, and verifying whether the resulting values are consistent with the distinguisher $\mathcal{D}$. If they are not, the candidate can be discarded. A central question is: for a given distinguisher $\mathcal{D}$ and the correct key k, which incorrect candidates $k_{\text{cand}} \neq k$ nevertheless pass this filtering step?

The dependence of the success of the key-recovery attack on the exact key is well documented in some classes of cryptanalysis. For linear cryptanalysis, the wrong-key randomisation hypothesis has been formalised and investigated [9][1], showing that the success probability can vary with the key, and that random behaviour only emerges under additional assumptions and averaging. For differential cryptanalysis, related phenomena have been observed: keys may split into equivalence classes that behave identically with respect to a given differential property [27], again contradicting a naive wrong-key model. The related notion of linear structures has been extensively used before, e.g in [16].

In this work, we focus on *integral distinguishers*. Concretely, we are given a set $M \subseteq \mathbb{F}_2^n$ and study divisibility properties of the sum of a fixed bit across all elements of M. The set M is described by its indicator function

$$\mu_M : \mathbb{F}_2^n \to \mathbb{Z},$$

where $\mu_M(x) = 1$ if $x \in M$ and $\mu_M(x) = 0$ otherwise. An integral distinguishing property $\mathcal{D} \coloneqq \mathcal{D}(M, \alpha, v)$ in the sense of this work is defined by a set $M \subseteq \mathbb{F}_2^n$, a fixed non-zero mask $\alpha \in \mathbb{F}_2^n$, and an integer $v \geq 2$; It asserts that the Fourier coefficient of μ_M in α is divisible by 2^v, i.e.,

$$\sum_{x \in \mathbb{F}_2^n} \mu_M(x)\,(-1)^{\langle \alpha, x \rangle} = \sum_{x \in M} (-1)^{\langle \alpha, x \rangle} \equiv 0 \mod 2^v.$$

Note that the set M itself arises from the encryption under $E'_{k'}$ of some initial set M' of plaintexts. In most attacks, M' is taken as an affine subspace of $\mathbb{F}_2^n$. Here, we allow any set M' of cardinality 2^m for m an integer greater than or equal to v.

Remark 1. The divisibility property of the sum of fixed bits across M is stated in a Fourier-like view above. Integral attacks typically work for sets of cardinality $|M| = 2^m$ with an integer $m \geq v$, e.g., with an input structure that is either a linear subspace or an affine shift thereof. In that case, we have

$$\sum_{x \in \mathbb{F}_2^n} \mu_M(x)(-1)^{\langle \alpha, x \rangle} \equiv 0 \mod 2^v$$

if and only if

$$\sum_{x\in\mathbb{F}_2^n} \mu_M(x)\cdot\langle x,\alpha\rangle = \sum_{x\in M}\langle \alpha, x\rangle \equiv 0 \mod 2^{v-1},$$

where the element $\langle\alpha, x\rangle$ is interpreted as an element in $\mathbb{Z}$. Note that we only obtain a meaningful distinguisher if $v > 1$. We will adopt the Fourier-like description in the remainder of this work.

Key-recovery in integral attacks. Denoting the key space of E by $\mathbb{F}_2^\kappa$, suppose that $k \in \mathbb{F}_2^\kappa$ is the correct key. By encrypting the initial set M' of plaintexts, an adversary observes $E_k(M)$ as the set of ciphertext outputs, but it does not know k. In the attack, the adversary would guess a key candidate $k_{\text{cand}} \coloneqq k + w$ (where w denotes the difference to the correct key) and computes backwards to obtain a set

$$M_{k,w} \coloneqq E_{k+w}^{-1} \circ E_k(M),$$

as illustrated in Figure 1. It would then check whether the property $\mathcal{D}(M_{k,w}, \alpha, v)$ holds for the obtained set $M_{k,w}$.

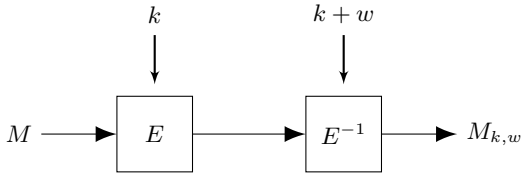


Fig. 1: Construction of $E_{k+w}^{-1} \circ E_k(M)$ occurring in the key-recovery phase for a key guess with difference w to the correct key

Up to now – most attacks do not even explicitly state it – the most-commonly used prevailing working assumption on the wrong-key behavior is Hypothesis 1. For the classical zero-sum case ($v = 2$), this heuristic is well known and widely used. For higher-order divisibility properties ($v \geq 3$), the same assumption is adopted as well, see e.g. [7]

Hypothesis 1 (Integral wrong key randomization hypothesis). *Let $k \in \mathbb{F}_2^\kappa$ be the correct key and M a set of size 2^m of elements in $\mathbb{F}_2^n$. For a non-zero $\alpha \in \mathbb{F}_2^n$ and $v \leq m$ a positive integer, suppose the integral distinguishing property $\mathcal{D}(M, \alpha, v)$ holds. Then the wrong key probability for the distinguisher to hold is*

$$Pr(\mathcal{D}(M_{k,w}, \alpha, v) \textit{ holds} \mid k, w \in \mathbb{F}_2^\kappa) = 2^{-v+1}. \tag{1}$$

Subsequently, the distribution of the total number of "surviving" wrong keys among 2^l guessed keys follows a binomial distribution

$$A \sim \text{Binomial}(2^l - 1, 2^{-v+1}).$$

As we already outlined in the introduction, there are two parts where this hypothesis can and will go wrong: i) key-clustering that results in a significantly larger variance, and ii) the fact the probability that a random subset yields divisibility by 2^{-v+1} is not exactly as expected. We will start by elaborating on the second fact in Section 3 and come back to the key-clustering in Section 4.

3 Wrong Key Behavior in Integral Attacks

In this section, we establish the theory of integral wrong-key randomization, compare it with previous hypotheses, and investigate the influence of fixed-key differences on the preservation of integral properties. We first focus, in Section 3.1, on the case where partial decryption behaves as an ideal random permutation. Even in this case, Hypothesis 1 cannot be fully justified. We then consider the case of non-ideal partial decryption and relate the statistical behavior to the differential-linear connectivity table in Section 3.2. Notably, for most applications with large subsets, it is, in our view, sufficient to rely on the generally accepted assumption – Hypothesis 1.

3.1 The behavior of $\mathcal{D}(M, \alpha, v)$ for randomly-chosen M

For now, we analyze the abstraction where no key, or key guess, is involved at all. In other words, we analyze the probability that the distinguishing property $\mathcal{D}(M, \alpha, v)$ holds for a uniformly random choice of M of a fixed cardinality. This model of abstraction ignores how the difference from the correct key affects the partial decryption through E.

A statement consistent with Hypothesis 1 would assert that such a random subset M of cardinality 2^m satisfies the property $\mathcal{D}(M, \alpha, v)$ for $v \leq m$ with probability 2^{-v+1}, which is contradicted by the next example.

Example 1. Let $M \subseteq \mathbb{F}_2^n = \mathbb{F}_2^4, |M| = 2^m = 8, v = 3, \alpha = 1$. For every element in M, we look only at the least-significant bit (LSB) (or, more generally, any non-zero linear mask α). We check the Fourier sum over all LSBs b_i in M, obtaining

$$S = \sum_{x \in \mathbb{F}_2^n} \mu_M(x) \, (-1)^{\langle \alpha, x \rangle} = \sum_{i=1}^{|M|} (-1)^{b_i} = \sum_{i=1}^{|M|} (1 - 2b_i) = |M| - 2 \sum_{i=1}^{|M|} b_i$$

with $b_i \in \{0, 1\}$ for $i = 1, \ldots, |M|$. Therefore,

$$S \equiv 0 \pmod 8 \iff \sum_{i=1}^{|M|} b_i \equiv 0 \pmod 4.$$

As M is chosen uniformly at random among all subsets of $\mathbb{F}_2^4$ of size $|M| = 8$, we are drawing 8 samples without replacement from a population of size $2^n = 16$, in which exactly half (i.e. 8) have LSB one, half have LSB zero. So, in order to

satisfy the condition on divisibility, among the LSBs of M there should be either zero 1's and eight 0's, or eight 1's and no 0's, or four 1's and four 0's. To obtain the probability for $\mathcal{D}(M, \alpha, v)$ to hold, the total number of such "satisfiable" sets should be normalized by all possible M. Hence,

$$\Pr\big(S \equiv 0 \bmod 2^v\big) = \frac{\binom{8}{0}\binom{8}{8} + \binom{8}{4}\binom{8}{4} + \binom{8}{8}\binom{8}{0}}{\binom{16}{8}} = \frac{817}{2145} \approx 0.38.$$

For comparison, the usual heuristic from Hypothesis 1 assumes the probability to be $2^{-v+1} = 2^{-2} = \frac{1}{4} = 0.25$.

The discrepancy arises because, firstly, M uses only a finite number of texts. Secondly, it is not taken into account that M is chosen *without replacement* from $\mathbb{F}_2^n$. Also, an important aspect is the exact value of v.

Computing the exact probability that $\mathcal{D}(M, \alpha, v)$ holds is a classical sampling without replacement problem: We draw $|M| = 2^m$ elements uniformly at random from a population of size 2^n containing exactly 2^{n-1} zeros and 2^{n-1} ones, so the count of ones follows a *hypergeometric*, not binomial distribution.

Theorem 1 (Distinguishing probability over random subset). *Let $m, v \in \mathbb{Z}$ with $m \geq v \geq 1$, $\alpha \in \mathbb{F}_2^n \setminus \{0\}$, and $M \subseteq \mathbb{F}_2^n$ be a subset of size 2^m chosen uniformly random. The probability for $\mathcal{D}(M, \alpha, v)$ to hold is*

$$\tau(n, m, v) \coloneqq \frac{1}{2^{v-1}} + \frac{1}{2^{v-1}\binom{2^n}{2^m}} \sum_{j=1}^{2^{v-1}-1} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} \omega^{js},$$

where $\omega = \exp(\frac{2\pi i}{2^{v-1}})$ and $i^2 = -1$.

Proof. We need to compute the probability that, given the tuple $(b_1, \ldots, b_{2^m}) \in \{0,1\}^{2^m}$, the sum $\sum_{i=1}^{2^m} b_i$ is divisible by 2^{v-1}. As mentioned, the sum follows the hypergeometric distribution,

$$X = \sum_{i=1}^{2^m} b_i \sim \text{Hypergeometric}(2^n, 2^{n-1}, 2^m),$$

where the probability mass function is given by $\Pr(X = x) = \frac{\binom{2^{n-1}}{x}\binom{2^{n-1}}{2^m - x}}{\binom{2^n}{2^m}}$.

The probability of X being divisible by 2^{v-1} is then obtained as

$$\begin{aligned}\Pr(X \equiv 0 \bmod 2^{v-1}) &= \sum_{s=0}^{2^m} \Pr(X = s) \cdot \mathbf{1}_{\,s \equiv 0 \pmod{2^{v-1}}} \\ &= \frac{1}{\binom{2^n}{2^m}} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} \mathbf{1}_{\,s \equiv 0 \pmod{2^{v-1}}}\end{aligned}$$

where $\mathbf{1}_{\,s \equiv 0 \pmod{2^{v-1}}}$ evaluates to 1 if s is divisible by 2^{v-1} and to 0 otherwise. To get a closed form, we use the transformation of the indicator to the

root-of-unity filter (see [32]): Denoting by $\omega = \exp(\frac{2\pi i}{2^{v-1}})$ a primitive 2^{v-1}-th root of unity, we have

$$\mathbf{1}_{s \equiv 0 \pmod{2^{v-1}}} = \frac{1}{2^{v-1}} \sum_{j=0}^{2^{v-1}-1} \omega^{js}.$$

With this, we obtain

$$\Pr(X \equiv 0 \bmod 2^{v-1}) = \frac{1}{2^{v-1}\binom{2^n}{2^m}} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} \sum_{j=0}^{2^{v-1}-1} \omega^{js}.$$

Then by isolating the term for $j = 0$ (corresponding to $\omega^0 = 1$) and Vandermonde's identity, we finally get

$$\begin{aligned}\Pr(X \equiv 0 \bmod 2^{v-1}) &= \frac{1}{2^{v-1}\binom{2^n}{2^m}} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} \\ &\quad + \frac{1}{2^{v-1}\binom{2^n}{2^m}} \sum_{j=1}^{2^{v-1}-1} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} \omega^{js} \\ &= \frac{1}{2^{v-1}} + \frac{1}{2^{v-1}\binom{2^n}{2^m}} \sum_{j=1}^{2^{v-1}-1} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} \omega^{js}.\end{aligned}$$

□

This probability is the "baseline" for a wrong key to be non-filtered on property $\mathcal{D}(M, \alpha, v)$ after key recovery. And its right-hand side is precisely the reason for the shifted mean depicted in the figures in the introduction. However, the probability also depends on how the difference from the correct key affects the partial decryption function, as discussed in the following subsection. Note that, for small m, the hypergeometric distribution can be well approximated by the binomial distribution. In this case, we could assume $b_i \sim \text{Bernoulli}(\frac{1}{2})$ for all $i = 1, \ldots, 2^m$.

We now look at the important special case of an integral property corresponding to $v = 2$ in more detail. The following corollary enables us to compute the probabilities in this case in a simpler way.

Corollary 1 (Probability for zero-sum property). *Let $\alpha \in \mathbb{F}_2^n \setminus \{0\}$ and $m \geq 2$. If $v = 2$, then probability of a random subset $M \subseteq \mathbb{F}_2^n$ of size 2^m fulfilling $\mathcal{D}(M, \alpha, v)$ is*

$$\tau(n, m, 2) = \frac{1}{2} + \frac{1}{2} \cdot \frac{\binom{2^{n-1}}{2^{m-1}}}{\binom{2^n}{2^m}}.$$

Proof. From Theorem 1, we have $\tau(n, m, 2) := \frac{1}{2} + \frac{1}{2\binom{2^n}{2^m}} \sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} (-1)^s$. The result then follows from the fact that

$$\sum_{s=0}^{2^m} \binom{2^{n-1}}{s} \binom{2^{n-1}}{2^m - s} (-1)^s = \binom{2^{n-1}}{2^{m-1}},$$

i.e., Equation 1.21 from [13]. □

For block sizes n used in practice, the second term becomes negligible, as long as $m < n$, and the probability coincides with the general assumed probability $\frac{1}{2}$. For example, $\tau(64, 4, 2) \approx 0.5 + 2^{-492.05}$ and $\tau(64, 63, 2) \approx 0.5 + 2^{-2^{63}}$.

More generally, for any fixed $v \ll n$, we expect the correction (second) term in $\tau(n, m, v)$ to decay rapidly as either the gap $m - v$ grows or as m approaches $n - 1$. For instance, $\tau(64, 4, 4) \approx 0.196$, while $\tau(64, 8, 4) \approx 0.125 + 2^{-31.24}$ and $\tau(64, 12, 4) \approx 0.125 + 2^{-469.86}$, so the residual "tail" is practically undetectable. In the typical use cases with integral distinguishers, $\tau(n, m, v)$ can be well approximated by the wrong-key heuristic $2^{-(v-1)}$ from Hypothesis 1.

3.2 Effect of key difference at wrong key randomisation

To illustrate how the baseline probability from Theorem 1 performs in practice, we use an integral distinguisher [34] on 7 rounds of `SKINNY-64` [6]. In this case, the distinguisher provides a balanced-sum property (i.e., $\mathcal{D}(M, \alpha, v)$ for $v = m$ and $|M| = 2^m$ with $m = 4$ and $n = 64$). The simple key-recovery experiment allows us to validate the theoretical baseline probabilities. Let M' be a subset of 2^4 plaintexts that iterates through all values in cell 15 while the remaining 15 input cells are fixed. Propagating M' through 7 rounds gives a balanced nibble sum on the output state: $S_7[6] \oplus S_7[14]$ (Fig. 2) over the sum of all texts in M'.

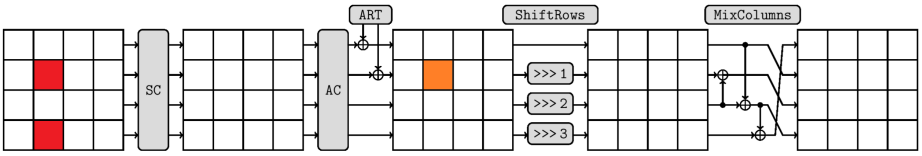


Fig. 2: 8-round key recovery on `SKINNY-64` using 7-round integral distinguisher (red color squares depict balanced sum property nibbles, whereas the orange one depicts the position of the guessed key nibble).

For the key recovery part, we attack an 8-round variant and guess only the sixth nibble of the sub-key, $K_8[6]$, and check balancedness for just the *LSB of the balanced nibble* (later, we consider combining properties from each bit). The property would hold only with probability $2^{-3} = 0.125$ per random trial for a wrong key if we use Hypothesis 1.

We conducted 10,000 instances of the key-recovery attacks, each one with a randomized correct key and a randomized constant part of the input. By investigating probabilities of $\mathcal{D}(M_{k,w}, \alpha, v)$ to hold for each difference w between the surviving wrong key nibble and the correct one k, Figure 3 exhibits a confirmation of Theorem 1: the majority exhibit the probability $\hat{p} \approx 0.196 = \tau(64, 4, 4)$, but for the XOR differences $w \in \{\texttt{0x5}, \texttt{0x6}, \texttt{0x9}, \texttt{0xA}, \texttt{0xC}, \texttt{0xF}\}$, we obtain slightly higher values. That can be explained as follows.

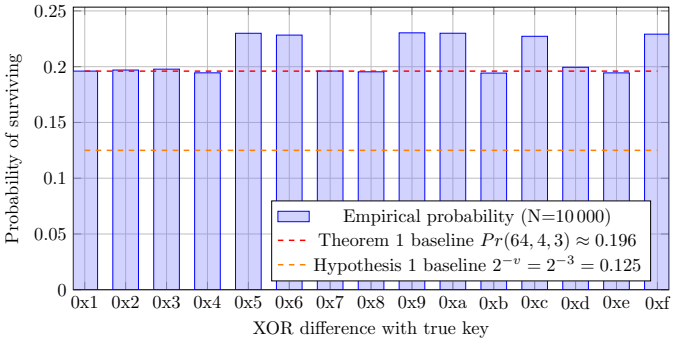


Fig. 3: Empirical (10,000 trials) survival probability for each wrong key in the 8-round SKINNY key-recovery experiment with balanced property *only on LSB*.

Taking $\tau(n, m, v)$ as the probability assumes that every wrong key guess behaves in the same way with respect to the integral bit, so the divisibility property $\mathcal{D}(M, \alpha, v)$ is satisfied with the baseline probability $\tau(n, m, v)$. In reality, when guessing a wrong key, we effectively impose a fixed XOR-difference w with respect to the correct key. This difference propagates backwards and modifies the partial decryption values, producing a new set $M_{k,w}$, whose component α is flipped with probability p, determined by the differential–linear effect of w.

Phrased differently, instead of the true subset M for the correct key (which satisfies $\mathcal{D}(M, \alpha, v)$), the wrong key forces us to analyse $M_{k,w}$, in which each element of the α–component has been flipped independently with probability p. Note that this assumption that each bit flips independently with some probability is slightly simplifying. The exact model would be to assume that the number of flipped bits follows a hypergeometric distribution with parameters $(2^n, p2^n, |M|)$. Here, we use the approximation by taking each bit flip independently from a Bernoulli distribution with parameter p.

Theorem 2 (Distinguishing probability after flipping bits in subset). *Let $m, v \in \mathbb{Z}$ with $m \geq v \geq 2$, $\alpha \in \mathbb{F}_2^n \setminus \{0\}$, and $M \subseteq \mathbb{F}_2^n$ be a subset of size 2^m chosen uniformly random. Assume that $\mathcal{D}(M, \alpha, v)$ holds, i.e., we have*

$$S := \sum_{x \in \mathbb{F}_2^n} \mu_M(x)\,(-1)^{\langle \alpha, x \rangle} = \sum_{i=1}^{|M|} (-1)^{b_i} \equiv 0 \bmod 2^v.$$

Let $(t_i)_{1 \leq i \leq 2^m}$ be i.i.d. Bernoulli(p) *variables and denote $S' := \sum_{i=1}^{|M|} (-1)^{b_i + t_i}$. The conditional probability* $\Pr(S' \equiv 0 \bmod 2^v \mid S \equiv 0 \bmod 2^v)$ *is equal to*

$$\frac{\displaystyle\sum_{k=0}^{2^{m-v+1}} \binom{2^{n-1}}{k\,2^{v-1}} \binom{2^{n-1}}{2^m - k\,2^{v-1}} f_p(m, v, k\,2^{v-1})}{\displaystyle\sum_{k=0}^{2^{m-v+1}} \binom{2^{n-1}}{k\,2^{v-1}} \binom{2^{n-1}}{2^m - k\,2^{v-1}}},$$

where $f_p(m,v,a) = \Pr\big(S' \equiv 0 \bmod 2^v \mid a \text{ ones among all } b_i\big)$. *We further have*

$$f_p(m,v,a) = \frac{1}{2^{v-1}} \sum_{j=0}^{2^{v-1}-1} \big(1-p+p\omega^j\big)^{2^m-a} \big(p+(1-p)\omega^j\big)^a, \quad \omega = \exp(\frac{2\pi i}{2^{v-1}}).$$

Proof. Conditioning on $S \equiv 0 \pmod{2^v}$ restricts $\sum_{i=1}^{|M|} b_i$ to values

$$a = k\,2^{v-1}, \qquad 0 \le k \le 2^{m-v+1}.$$

Hence, the probability $\Pr(S' \equiv 0 \bmod 2^v \mid S \equiv 0 \bmod 2^v)$ can be expressed as

$$\sum_{k=0}^{2^{m-v+1}} \Pr\left(S' \equiv 0 \bmod 2^v \mid \sum_{i=1}^{2^m} b_i = k2^{v-1}\right) \Pr\left(\sum_{i=1}^{2^m} b_i = k2^{v-1} \mid S \equiv 0 \bmod 2^v\right),$$

which is equal to

$$\sum_{k=0}^{2^{m-v+1}} f_p\big(m,v,k\,2^{v-1}\big) \frac{\binom{2^{n-1}}{k\,2^{v-1}}\binom{2^{n-1}}{2^m-k\,2^{v-1}}}{\sum_{k'=0}^{2^{m-v+1}}\binom{2^{n-1}}{k'\,2^{v-1}}\binom{2^{n-1}}{2^m-k'\,2^{v-1}}}$$

with $f_p(m,v,a) = \Pr\big(S' \equiv 0 \bmod 2^v \mid a = \sum_{i=1}^{2^m} b_i\big)$, because $\sum_{i=1}^{2^m} b_i$ follows a hypergeometric distribution with parameters $2^n, 2^{n-1}, 2^m$.

It is left to analyze $f_p(m,v,a)$. The condition $\sum_{i=1}^{2^m} b_i = a$ means that there are $2^m - a$ zeros and a ones in $(b_1,\ldots,b_{2^m})$. After flipping each b_i with probability p, the sum $\sum_{i=1}^{2^m}(b_i \oplus t_i)$ corresponds to the sum of two random variables X and Y, where

$$X \sim \text{Binomial}\big(2^m - a,\, p\big) \quad \text{(zeros that flip to 1)},$$

$$Y \sim \text{Binomial}\big(a,\, 1-p\big) \quad \text{(ones that stay 1)}.$$

Evaluating the probabilities $\Pr[X+Y=s]$ through direct convolution is not practical, but the probability generating function (pgf) [15] turns sums of independent variables into products of generating functions, which is especially convenient once we apply the roots-of-unity filter for congruences modulo 2^{v-1}.

For a discrete random variable H taking non-negative integers, its pgf is defined as

$$G_H(z) \coloneqq \mathbb{E}[z^H] = \sum_{x=0}^{\infty} \Pr[H=x] \cdot z^x.$$

For $H \sim \text{Binomial}(N,q)$, we have $G_H(z) = (1-q+qz)^N$. In our case, we therefore have

$$G_X(\omega^j) = \mathbb{E}\left[\omega^{jX}\right] = (1-p+p\omega^j)^{2^m-a}, \quad G_Y(\omega^j) = \mathbb{E}\big[\omega^{jY}\big] = (p+(1-p)\omega^j)^a.$$

Independence yields

$$G_{X+Y}(\omega^j) = G_X(\omega^j)G_Y(\omega^j) = (1-p+p\omega^j)^{2^m-a}\,(p+(1-p)\omega^j)^a.$$

Using the roots-of-unity filter as in Theorem 1, we obtain

$$
\begin{aligned}
f_p(m,v,a) &= \sum_{s=0}^{2^m} \Pr[X+Y=s]\cdot \mathbf{1}_{s\equiv 0 \pmod{2^{v-1}}} = \mathbb{E}[\mathbf{1}_{X+Y\equiv 0 \pmod{2^{v-1}}}] \\
&= \mathbb{E}\left[\frac{1}{2^{v-1}}\sum_{j=0}^{2^{v-1}-1}\omega^{j(X+Y)}\right] = \frac{1}{2^{v-1}}\sum_{j=0}^{2^{v-1}-1}\mathbb{E}\left[\omega^{j(X+Y)}\right] \\
&= \frac{1}{2^{v-1}}\sum_{j=0}^{2^{v-1}-1}(1-p+p\omega^j)^{2^m-a}\,(p+(1-p)\omega^j)^a.
\end{aligned}
$$

□

Let us discuss the implications of this theorem: If $p = 0$, no bits flip, so $S' = S$ and the property survives with probability 1; when $p = 1$ every bit is complemented, but 2^m is itself a multiple of 2^v, so $S' = 2^m - S \equiv -S \equiv 0 \pmod{2^v}$ and the survival probability is again 1. For the balanced value $p = \frac{1}{2}$, the two factors in the sum expression in $f_p(m, v, a)$ coincide, the dependence on the specific weight a disappears, and the result diminishes to the baseline probability similarly as stated in Theorem 1, but assuming that $\sum_{i=1}^{2^m} b_i$ follows a Binomial distribution with parameters 2^m and $1/2$. More generally, exchanging p and $1-p$ swaps the exponents a and 2^m-a; because the conditional distribution of a is symmetric and both a and $2^m - a$ are multiples of 2^{v-1}, the averaged probability is unchanged. Consequently, $\Pr_p(S' \equiv 0 \bmod 2^v \mid S \equiv 0 \bmod 2^v) = \Pr_{1-p}(S' \equiv 0 \bmod 2^v \mid S \equiv 0 \bmod 2^v)$.

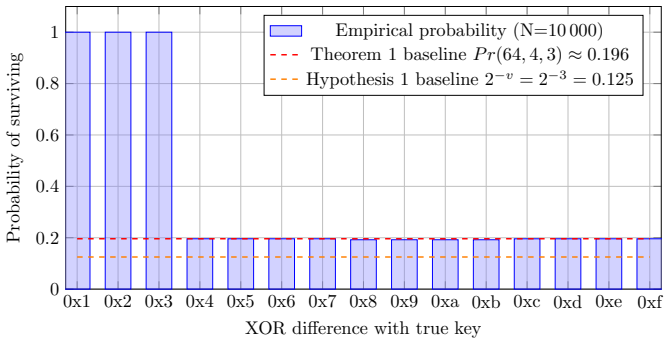


Fig. 4: Empirical (10,000 trials) survival probability for each wrong key in the 8-round SKINNY key-recovery experiment with balanced property *on the second LSB*.

Theorem 2 explains why the six differences with bias $\pm 1/4$ in the DLCT of the SKINNY inverse S-box yield a higher survival probability ($p \approx 0.229$, see

Table 2: Survival of key-nibble candidates after the 7-round LSB integral.

XOR diff d	**DLCT Bias** ε (α=`0x1`)	$p(d)$ (flip-prob.)	**Theorem 2 prob.**	**Observed prob.**
`0x0` (correct)	1/2	1	1.000	1.0000
`0x1`	0	1/2	0.196	0.1960
`0x2`	0	1/2	0.196	0.1932
`0x3`	0	1/2	0.196	0.1974
`0x4`	0	1/2	0.196	0.1981
`0x7`	0	1/2	0.196	0.1958
`0x8`	0	1/2	0.196	0.1990
`0xB`	0	1/2	0.196	0.1867
`0xD`	0	1/2	0.196	0.1957
`0xE`	0	1/2	0.196	0.1989
`0x5`	1/4	3/4	0.229	0.2358
`0x6`	1/4	3/4	0.229	0.2277
`0x9`	−1/4	1/4	0.229	0.2279
`0xA`	−1/4	1/4	0.229	0.2280
`0xC`	−1/4	1/4	0.229	0.2386
`0xF`	−1/4	1/4	0.229	0.2277

Table 2), whereas the other nine differences with zero bias follow the baseline prediction ($p \approx 0.196$), reflecting that the wrong-key distribution is governed by differential-linear effects, and these biases must be accounted for explicitly.

Remark 2. Note, when the DLCT bias equals $\pm 1/2$ (for a 4-bit S-box), the component is perfectly correlated: the induced probability is $p \in \{0, 1\}$ (see Fig. 4, which depicts the case of integral property on 2nd LSB). This exactly means the presence of a linear structure, so any wrong key with the corresponding XOR difference w will *always* preserve the integral property. Moreover, exactly this is the root of key clustering. We discuss this case in detail, and in the following section, where we formalize it and automate the detection.

Remark 3. All probabilities above were for an integral property on a *single* bit (LSB of the nibble). If we test several integral properties simultaneously (whole nibble, for example, see Fig. 5), the survival probability becomes a function of the probabilities per coordinate. In our `SKINNY` experiment, the last decryption step XORs two nibbles, one depending on the guessed key nibble and one not depending; this induces sufficient "randomization" that the four coordinates of the balanced nibble can be assumed to behave independently. Therefore, the final probability is a product of the per-coordinate probabilities.

Under Hypothesis 1, the filter for the full nibble would be $(2^{-3})^4 = 2^{-12} \approx 0.000244$. Fig. 5 shows this does not hold uniformly. For example, for XOR difference `0x2`, Theorem 2 gives per-bit survival probabilities (1, 0.196, 0.196, 1) for masks $(1, 2, 4, 8)$, so the combined probability is $1 \cdot 0.196 \cdot 0.196 \cdot 1 \approx 0.0384$,

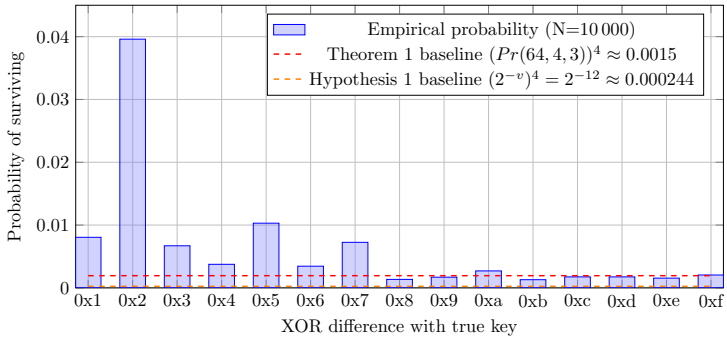


Fig. 5: Empirical (10,000 trials) survival probability for each wrong key in the 8-round SKINNY key-recovery experiment with balanced property *on whole nibble*.

matching the observed ≈ 0.04. Notably, in other ciphers, independence of coordinates does not necessarily hold – partial decryption functions may be correlated, yet the final distribution can still be driven by the differential-linear bias of the inverse Boolean function.

4 Improved Key-Guessing: Key Information that Matters

We now focus on the following question in more detail: for a given integral distinguisher $\mathcal{D}(M, \alpha, v)$ and the correct key k, which incorrect key candidates $k_{\text{cand}} \neq k$ pass the wrong-key filtering procedure, i.e., for which $k_{\text{cand}} = k + w$ do we have $\mathcal{D}(M_{k,w}, \alpha, v)$ assuming that $\mathcal{D}(M, \alpha, v)$ holds? Recall that

$$M_{k,w} = E_{k+w}^{-1} \circ E_k(M).$$

This question motivates the following definition:

Definition 1 ((M, α, v)-admissible key difference). *Let E be a cipher with key space $\mathbb{F}_2^\kappa$ and message space $\mathbb{F}_2^n$, and let $k \in \mathbb{F}_2^\kappa$. A key difference $w \in \mathbb{F}_2^\kappa$ is* (M, α, v)-admissible for (E, k) *if*

$$\mathcal{D}(M, \alpha, v) \quad \Longleftrightarrow \quad \mathcal{D}(M_{k,w}, \alpha, v).$$

In other words, a candidate key $k_{\text{cand}} = k + w$ is considered indistinguishable from the correct one if it preserves the integral distinguisher defined by (M, α, v) after partial decryption of the ciphertexts. Note that the (M, α, v)-admissibility not only depends on the cipher part E, but also on the correct key k that is used for encryption.

Computing the set of surviving key candidates as defined above is, in general, infeasible. We therefore impose the following simplifications to our model.

1. Instead of evaluating divisibility conditions over sums in the distinguisher, we impose the stronger requirement that, for a fixed mask α, the exact values $\langle \alpha, x \rangle$ are preserved under the key guess k_{cand} (or they are always flipped).

2. The admissibility of a key difference should be independent of the correct key k used in the encryption, leading to a more robust notion in practice.

This is captured in the following notion:

Definition 2 (α-admissible key differences). *Let E be a cipher with key space $\mathbb{F}_2^\kappa$ and message space $\mathbb{F}_2^n$. A key difference $w \in \mathbb{F}_2^\kappa$ is called* α-admissible for (E, k) *if the function*

$$x \mapsto \langle \alpha, E_k^{-1}(x) \rangle + \langle \alpha, E_{k+w}^{-1}(x) \rangle$$

is constant on $\mathbb{F}_2^n$. A key difference $w \in \mathbb{F}_2^\kappa$ is called α-admissible for E *if it is α-admissible for all (E, k) for all $k \in \mathbb{F}_2^\kappa$.*

Phrased differently, w is α-admissible for E if and only if w is a linear structure in the key space of the component function $(x, k) \mapsto \langle \alpha, E_k^{-1}(x) \rangle$. Note, a *linear structure* for a Boolean function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2$ is defined as an element $\delta \in \mathbb{F}_2^n$ for which the derivative $x \mapsto f(x) + f(x + \delta)$ is equal to a constant c. If $c = 0$ we say *LS-0*; if $c = 1$, we say *LS-1*.

For a fixed cipher E and mask α, it is apparent from this definition that the set of α-admissible key differences for E forms a vector space.

The following lemma, which follows directly from the definition, states that an α-admissible key difference preserves any integral distinguisher defined by α.

Lemma 1. *Fix any tuple (M, v, k). If w is α-admissible for (E, k), then w is (M, α, v)-admissible for (E, k).*

Improving the key-recovery phase when knowing α-admissible key differences. Given a cipher E and a non-zero mask α, our goal is to compute the space of α-admissible key differences for E. Suppose we have a subspace $W \subseteq \mathbb{F}_2^\kappa$ of those. Then, for each $w \in W$, a candidate key $k + w$ is indistinguishable from the candidate key k. In other words, all keys in the coset $k + W$ behave the same way within the filtering phase. To recover the "correct" coset, one therefore only has to iterate through key guesses k_{cand} from the quotient space $\mathbb{F}_2^\kappa / W$. The complexity to do so then reduces from 2^κ (i.e., the case where all keys from $\mathbb{F}_2^\kappa$ need to be guessed) to $2^{\kappa - \dim W}$.

4.1 Finding Linear Structures in the Key: Affine Related-Key Subspace Trails (ARKST)

From now on, we assume that E is a key-alternating cipher with independent round keys. That is, $k = (k_1, \ldots, k_r)$ with each $k_i \in \mathbb{F}_2^n$, and

$$E_k = \mathrm{Add}_{k_r} \circ R \circ \cdots \circ \mathrm{Add}_{k_2} \circ R \circ \mathrm{Add}_{k_1} \circ R$$

for a bijective round function $R\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$.

The case of one round. As the simplest example, we consider a key-recovery over a single round, i.e., E is a 1-round key-alternating cipher so that

$$E_k(x) = R(x) + k.$$

Then, $x \mapsto \langle \alpha, E_k^{-1}(x)\rangle + \langle \alpha, E_{k+w}^{-1}(x)\rangle$ simplifies to $x \mapsto \langle \alpha, R^{-1}(x+k)\rangle + \langle \alpha, R^{-1}(x+k+w)\rangle$ and the question whether this is a constant function on $\mathbb{F}_2^n$ is independent of k. More precisely, the following three statements are equivalent:

1. w is α-admissible for E.
2. w is α-admissible for (E, k) for an arbitrary fixed key k.
3. w is a linear structure of the component function $\langle \alpha, R^{-1}\rangle$.

The case of multiple rounds. Our goal is to characterize and to find α-admissible key differences for E. We will use the following notation, coming from the theory of subspace trails [14,26].

Definition 3. *Let $F\colon \mathbb{F}_2^m \to \mathbb{F}_2^n$ be a function and let $U \subseteq \mathbb{F}_2^m, V \subseteq \mathbb{F}_2^n$ be non-empty sets. We write*

$$U \xrightarrow{F} V$$

if and only if $\forall x \in \mathbb{F}_2^m, \forall u \in U\colon F(x) + F(x+u) \in V$.

Using this notation, w is α-admissible for E if and only if there exists a set $V \subseteq \mathbb{F}_2^n$ such that $\langle \alpha, x\rangle$ is constant on all $x \in V$ and for which the propagation $\{0\} \times \{w\} \xrightarrow{E^{-1}} V$ holds. Finding such w for our case of a key-alternating cipher will be done in a trail-based manner. More precisely, we use the following propagation rule:

Lemma 2 (Propagation through key-alternating cipher). *Let*

$$E^{-1}\colon \mathbb{F}_2^n \times (\mathbb{F}_2^n)^r \to \mathbb{F}_2^n, (x, (k_1, \ldots, k_r)) \mapsto R_{k_1}^{-1} \circ R_{k_2}^{-1} \circ \cdots \circ R_{k_{r-1}}^{-1} \circ R_{k_r}^{-1}(x)$$

with $R_{k_i}^{-1}(x) \coloneqq R^{-1}(x + k_i)$. For $i = 1, \ldots r$, let $V_i + W_i \xrightarrow{R^{-1}} V_{i-1}$ hold for sets $V_i, W_i, V_0 \subseteq \mathbb{F}_2^n$. Then,

$$V_r \times (W_1 \times W_2 \times \cdots \times W_r) \xrightarrow{E^{-1}} V_0.$$

This means that $w = (w_1, \ldots, w_r)$ is α-admissible for E if there is a sequence $V_r, V_{r-1}, \ldots, V_1, V_0$ of subsets of $\mathbb{F}_2^n$ with $V_r = \{0\}$, $\langle \alpha, x\rangle$ being constant for $x \in V_0$, and $V_i + w_i \xrightarrow{R^{-1}} V_{i-1}$ for all $i = 1, \ldots, r$.

Running through all w and checking its α-admissibility for E is not feasible for ciphers used in practice. Our idea for finding α-admissible w is to extend a given prefix $(w_1, \ldots, w_k)$ gradually by guessing a promising candidate for w_{k+1}, leading to an extension $(w_1, \ldots, w_r)$ that is α-admissible. To do so, we restrict to the the case where the sets V_i are affine subspaces of $\mathbb{F}_2^n$.

An *affine space* $U \subseteq \mathbb{F}_2^n$ can be written as $U^{\text{lin}} + u$, where U^{lin} is a vector subspace of $\mathbb{F}_2^n$ and $u \in \mathbb{F}_2^n$. Given a non-empty set $X \in \mathbb{F}_2^n$, we denote by $\text{aff}(X)$ the *affine span* of X, i.e., the intersection of all affine spaces of $\mathbb{F}_2^n$ containing X. For arbitrary $x \in X$, we have $\text{aff}(X) = x + \text{span}(X + x)$.

In fact, we conjecture that any α-admissible key difference induces an affine subspace trail and thus computing all those subspace trails ensures that all α-admissible key differences are captured. We would also like to highlight already now that studying affine subspaces is crucial for the algorithmic approach below as it allows us to consider zero-dimensional W_i.

4.2 The ARKST-Algorithm

For computing the propagation of affine spaces, we use the following lemma, which states that in order to propagate an affine space, it is enough to propagate elements corresponding to a basis, just like it was shown for the non-affine case in [26]. Note that, in some sense, our approach can be considered dual to linear mask propagation in [25].

Lemma 3 (Evaluating propagations for affine spaces). *Let* $F\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ *be a function and let* $U = U^{\text{lin}} + u \subseteq \mathbb{F}_2^n$ *be an affine space with* $U^{\text{lin}} = \text{span}\{u_1, \ldots, u_d\}$. *Let*

$$\mathcal{O} \coloneqq \{F(x) + F(x+u) \mid x \in \mathbb{F}_2^n\}, \quad \mathcal{B}_i \coloneqq \{F(x) + F(x+u_i+u) \mid x \in \mathbb{F}_2^n\}.$$

Then, we have the propagation $U \xrightarrow{F} \text{aff}(\mathcal{O} \cup \bigcup_{i=1}^d \mathcal{B}_i)$. *Moreover, any affine space* $V \subseteq \mathbb{F}_2^n$ *for which* $U \xrightarrow{F} V$ *holds must contain* $\text{aff}(\mathcal{O} \cup \bigcup_{i=1}^d \mathcal{B}_i)$.

Proof. Let us take an element $\sum_{i \in S} u_i + u \in U$ for an index set $S \subseteq \{1, \ldots, d\}$. By induction on $|S|$, we observe that for any $x \in \mathbb{F}_2^n$, we can find elements $b_i \in \mathcal{B}_i$ such that

$$F(x) + F(x + \sum_{i \in S} u_i + u) = (|S| + 1) \cdot o_x + \sum_{i \in S} b_i, \tag{2}$$

where $o_x \coloneqq F(x) + F(x+u) \in \mathcal{O}$. Indeed, for $|S| = 0$, the statement is trivial. For the inductive step, taking $j \notin S$, observe that

$$\begin{aligned} &F(x) + F(x + u_j + \sum_{i \in S} u_i + u) \\ =&F(x) + F(x + u_j) + F(x + u_j) + F(x + u_j + \sum_{i \in S} u_i + u) \\ =&F(x) + F(x + u_j) + (|S| + 1) \cdot o_{x+u_j} + \sum_{i \in S} b_i \end{aligned}$$

for elements $b_i \in \mathcal{B}_i$, where the last equality holds from the induction hypothesis. We can further rewrite $F(x) + F(x + u_j)$ as

$$F(x) + F(x + u_j + u) + F(x + u_j) + F(x + u_j + u) = b_j + o_{x+u_j}$$

for an element $b_j \in \mathcal{B}_j$, but we can also express $F(x) + F(x + u_j)$ as

$$F(x) + F(x + u) + F(x + u_j) + F(x + u) = o_x + b'_j$$

for an element $b'_j \in \mathcal{B}_j$. The claim now follows by expressing $F(x) + F(x + u_j)$ in the first form in case that $|S|$ is even, and expressing $F(x) + F(x + u_j)$ in the second form in case that $|S|$ is odd.

We see that every sum of the form as in (2) is contained in $\mathrm{aff}(\mathcal{O} \cup \bigcup_{i=1}^{d} \mathcal{B}_i)$. Indeed, if $|S|$ is even, then

$$F(x) + F(x + \sum_{i \in S} u_i + u) = o_x + \sum_{i \in S} b_i = o_x + \sum_{i \in S} (b_i + o_x).$$

If $|S|$ is odd, then

$$F(x) + F(x + \sum_{i \in S} u_i + u) = \sum_{i \in S} b_i = o_x + \sum_{i \in S} (b_i + o_x).$$

In both cases, the element lies in $o_x + \mathrm{span}(\bigcup_{i=1}^{d} (\mathcal{B}_i + o_x))$.

Let now $V \subseteq \mathbb{F}_2^n$ be an affine space for which the propagation $U \xrightarrow{F} V$ holds. By definition of the propagation, V must contain all elements from $\mathcal{O}$ and all elements from $\mathcal{B}_i$ for $i = 1, \ldots, d$ (propagate all elements from U for $|S| \leq 1$). The result follows, because $\mathrm{aff}(\mathcal{O} \cup \bigcup_{i=1}^{d} \mathcal{B}_i)$ is defined to be the smallest affine space containing $\mathcal{O} \cup \bigcup_{i=1}^{d} \mathcal{B}_i$. □

For an exact computation of the propagation of an affine space U over a function F, we would need to evaluate $F(x) + F(x + u)$ and $F(x) + F(x + u_i)$ for all $i \in \{1, \ldots, d\}$ and all $x \in \mathbb{F}_2^n$. Depending on the structure of U, this can be infeasible due to the huge block size n used in practical ciphers. To overcome this issue, we will compute the elements of $\mathcal{O}$ and $\mathcal{B}_i$, $i = 1, \ldots, d$ for a uniformly random sample of elements $x \in \mathbb{F}_2^n$ of size Z and compute the affine span only over those elements. Note that the error probability (i.e., the probability of not hitting the whole space $\mathrm{aff}(\mathcal{O} \cup \bigcup_{i=1}^{d} \mathcal{B}_i)$ from the samples) decreases exponentially with Z, assuming the average case when the set behaves randomly. To keep the probability of error negligible, we suggest using $Z = 50$.

The case of an SPN. We now study the case where the cipher E is an SPN, i.e., the round R can be described as $R = L \circ \mathbb{S}$ with $\mathbb{S}$ denoting a parallel S-box layer of bijective S-boxes $S \colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ and $L \colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ a linear bijection. Let us denote by t the number of parallel S-boxes, i.e., $n = tb$.

Suppose we have an affine space $V_i = V_i^{\mathrm{lin}} + v_i$ with V_i^{lin} being a linear subspace of $\mathbb{F}_2^n$. Our algorithm for finding an α-admissible key differences $w = (w_1, \ldots, w_r)$ for E will choose a zero-dimensional affine space $W_i = \{w_i\}$, and then computes an affine space V_{i-1} such that the propagation $U_i \xrightarrow{R^{-1}} V_{i-1}$ holds, where $U_i \coloneqq V_i + W_i$.

Sticking to zero-dimensional W_i eventually captures all the cases by taking the span of those, as linear structures form a subspace. However, even the number of zero-dimensional W_i is too large in practice to exhaust all in a reasonable time.

From the previous lemma, we take $V_{i-1} = \mathrm{aff}(\mathcal{O}_i \cup \bigcup_{j=1}^{d_i} \mathcal{B}_{i,j})$, where

$$\mathcal{O}_i \coloneqq \{F(x) + F(x + u_i) \mid x \in \mathbb{F}_2^n\}, \quad \mathcal{B}_{i,j} \coloneqq \{F(x) + F(x + u_{i,j} + u) \mid x \in \mathbb{F}_2^n\}$$

and $U_i = \mathrm{span}(u_{i,1}, \ldots, u_{i,d_i}) + u_i$. To keep V_{i-1} as small as possible, the idea is that U_i activates only a few S-boxes. To do so, we will only take w_i of the form $v_i + a_i$, where a_i activates at most one S-box. Our algorithm will iterate over all a_i of such form, i.e., for each round, it iterates over $t(2^b - 1) + 1$ choices for a_i. Because a_i should activate only one S-box, we have $a_i = L(c_i)$ for $\mathrm{wt}_b(c_i) \leq 1$, where $\mathrm{wt}_b(c_i)$ denotes the number of non-zero b-bit words in the vector c_i.

We conjecture that using those W_i, activating only one S-box at a time, we capture all existing α-admissible key differences. That is not to say that all admissible key differences are of this form, but that all of them can be expressed as linear combinations of those minimal ones. We list the algorithm in pseudocode in Algorithm 1.

We demonstrate this approach on the best-known attacks for `PRESENT`, `GIFT`, and `RECTANGLE`, all of which use bit-permutation linear layers. A key advantage of our framework is that it requires only a description of the linear layer and the S-box to automatically compute the neutral-bit subspace. This makes our tool generic.

5 Applications

We applied Algorithm 1 for the ciphers listed in Table 1. It returned spaces of α-admissible key differences for the relevant masks α. Due to our conjecture that the ARKST approach finds the complete space of linear structures in the key, we do not expect that there are any other α-admissible key differences.

These spaces correspond to key information bits that do not influence whether a given divisibility property holds after partial decryption. Importantly, many of these bits were previously assumed to affect the integral property and were thus included in the key-guessing phase of attacks. Obviously, excluding them from guessing directly would reduce the time complexity of the integral key-recovery attacks.

A strength of our approach is that it automatically detects non-trivial relations between key bits. For example, if flipping one subkey bit from the correct one can be compensated for by flipping some other bits (possibly in a different round), the joint effect leaves the integral property invariant and thus yields an α-admissible key difference. Algebraically, such dependencies appear as non-unit vectors in the row-echelon basis of the admissible key difference space W returned by Algorithm 1; concrete bases for W are listed in Supplementary Material B.

All key-recovery time complexity improvements are reported in Table 1. The applied techniques are identical to those in the original works for fair comparison.

Algorithm 1 Finding α-admissible key differences for SPNs (ARKST)

$L\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ being the linear layer of the round R
$\mathbb{S}$ being the S-box layer of the round R
$F\colon \mathbb{F}_2^n \to \mathbb{F}_2^n, F(x) = \mathbb{S}^{-1} \circ L^{-1}(x)$ being the inverse round R^{-1}
r being the number of rounds
b being the input/output size of the S-box

$R \leftarrow \{\}$ ▷ Stores the α-admissible key differences found
procedure PROPAGATE($i, V_i, [w_r, w_{r-1}, \ldots, w_{i+1}], \alpha$) ▷ Recursive
  if $i = 0$ **then**
    if $\langle \alpha, x \rangle$ is constant on $x \in V_0$ **then** ▷ Check by Z random samples
      $R \leftarrow R \cup \{(w_1, w_2, \ldots, w_r)\}$
    end if
  else
    for all c_i with $\mathrm{wt}_b(c_i) \leq 1$ **do** ▷ Activate only one S-box
      $w_i \coloneqq v_i + L(c_i)$
      $U_i \coloneqq V_i + w_i$
      Compute $V_{i-1} \coloneqq \mathrm{aff}(\mathcal{O}_i \cup \bigcup_{j=1}^{d_i} \mathcal{B}_{i,j})$ ▷ using Z random samples
      PROPAGATE($i-1, V_{i-1}, [w_r, w_{r-1}, \ldots, w_i], \alpha$)
    end for
  end if
end procedure

procedure FINDADMISSIBLE(r, α)
  $V_r \coloneqq \{0\}$
  PROPAGATE($r, V_r, [], \alpha$)
end procedure

FINDADMISSIBLE(r, α)
$R \leftarrow \mathrm{span}(R)$ ▷ The space of α-admissible key differences found

The only modification is the reduced number of key information to be guessed, which leads to substantial improvements over the state-of-the-art.

Two most widely optimisation techniques used in integral key recovery for zero-sum (mod 2) – FFT and partial-sums, stay essentially unchanged in the divisibility setting (mod 2^v).

Fast Fourier Transform (FFT) key-recovery [29]. Many integral key-recovery exploits FFT to evaluate the sum of partial decryptions

$$\bigoplus_{c \in E_k \circ E'_{k'}(M')} f_{k_1}(c \oplus k_2)\,,$$

where the bit length of k_1 and k_2 are κ_1 and κ_2, respectively. Then, by using the fast Walsh–Hadamard transform (FWHT), all XORs are computed with time complexity $3 \times \kappa_2 \times 2^{\kappa_1+\kappa_2}$ additions (independent of the number of chosen plaintexts) instead of $2^{\kappa_1+\kappa_2} \times |M|$ Boolean function evaluations. As FWHT used

in the mod-2 case already operates with integer additions/subtractions, the final reduction modulo 2 is applied only after the transform. Thus, complexity remains the same independently of the value of v.

Partial sums key-recovery [12]. Partial sums precompute and store the per-plaintext contribution for small subkey slices. When extending a key guess, we combine the cached values instead of recomputing over all texts. The only change is the type of the counter: instead of a single parity bit, store an integer counter reduced modulo 2^v. Hence, the time complexity is unchanged; the memory per bucket grows from 1 bit to v bits (i.e., in total by a factor of $v - 1$).

Beginning from the fourth key-recovery round onward, our ARKST search becomes computationally challenging. This is fine in practice: after the round function reaches full diffusion, the subspace of α-admissible key differences does not grow any further. Therefore, it is enough to search up to the first full diffusion depth – three rounds for `PRESENT` and `GIFT`, and four rounds for `RECTANGLE`. The analysis of 1-2 rounds instances can be done in less than a second, for three rounds, results are obtained in less than 10 minutes, and for four rounds, execution finished in several days, all on a common PC.

All of the α-admissible key differences that our ARKST tool outputs can be easily verified by hand. This gives confidence that the reported subspaces are complete under our model and assumptions.

We also examined `SIMON32` to find α-admissible key differences (see Supplementary Material A). Although we did not apply Algorithm 1, our analysis shows that there exist bits whose combinations also do not impact the distinguishing condition, confirming that the phenomenon is not limited to SPN ciphers with bit permutations.

Remark 4. Our method is applicable to any SPN in general, but, in our opinion, it is most promising for bit-wise/bit-slice ciphers (with a bit-permutation linear layer). Moreover, the method is applicable to many other cryptanalysis techniques. This can be anything where the distinguisher evaluates fixed bits of the internal state, for example, linear attacks.

5.1 Application to `PRESENT`

`PRESENT` is a lightweight block cipher introduced by Bogdanov et al. in 2007 [8], designed for resource-constrained devices. It comes in two variants: `PRESENT-80` with an 80-bit key and `PRESENT-128` with a 128-bit key. The cipher uses a simple round function, consisting of an S-box layer, a bit-permutation layer, and a key addition. To demonstrate our results, we will focus on 80-bit version.

As already discussed, an important detail on the `PRESENT` S-box is that its inverse has a linear structure on its LSB (mask $\alpha =$ `0x1`). For `PRESENT`, LS-0 corresponds to difference `0x5`, while LS-1 corresponds to `0x1` and `0x4`. This interacts strongly with integral distinguishers: Most integral distinguishers target the rightmost nibble of its state, and the subsequent key-recovery phase propagates

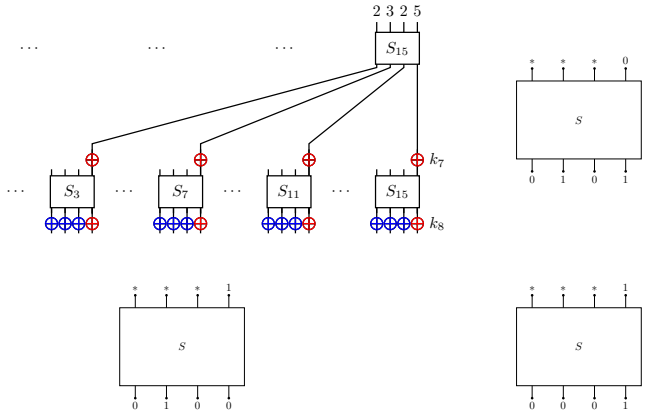


Fig. 6: **Top-left:** 2-round key recovery on `PRESENT` using a divisibility property on the least-significant nibble [7]; red key bits denote bits that, due to α-admissible key differences, do not affect whether the integral holds, while blue bits are the ones that actually should remain to be guessed. **Other:** linear structures (LS-0/LS-1 components) of the `PRESENT` inverse S-box.

to the whole state through LSB wires after each S-box. That makes `PRESENT` particularly suitable for applying ARKST to find a larger-dimensional subspace R in Algorithm 1.

Illustrative example. Firstly, let us discuss an 8-round attack (see Fig. 6) from the introduction to build our intuition. Assume that one of the bits in subkey k_7 is guessed wrongly. The error difference in this key bit guess can be canceled by a simultaneous wrong guess for key bits in the previous subkey. Namely, if we guess wrongly for the corresponding k_8 nibble with a difference of LS-1, then it will flip back the k_7 bit as if it were guessed correctly. On the other hand, if k_7 was guessed correctly, then any LS-0 difference in the corresponding k_8 nibble S-box would also yield a correct (because of propagated zero difference) value for the next step. In both cases, due to the structure of LS (which is always a subspace), each nibble of k_8 spans a subspace of dimension 3, meaning its LSB remains free. Additionally, this shows that an incorrect bit in k_7 does not change the distinguisher outcome, since it can always be compensated. Therefore, these two bits may not be considered during key recovery, as well as three more pairs. In total, the key recovery's 20-bit key subspace is reduced by a dimension of 8. These compensations are not ad-hoc: our tool automatically identifies them – the non-unit vectors in the basis of α-admissible key differences correspond exactly to such cross-round cancellations.

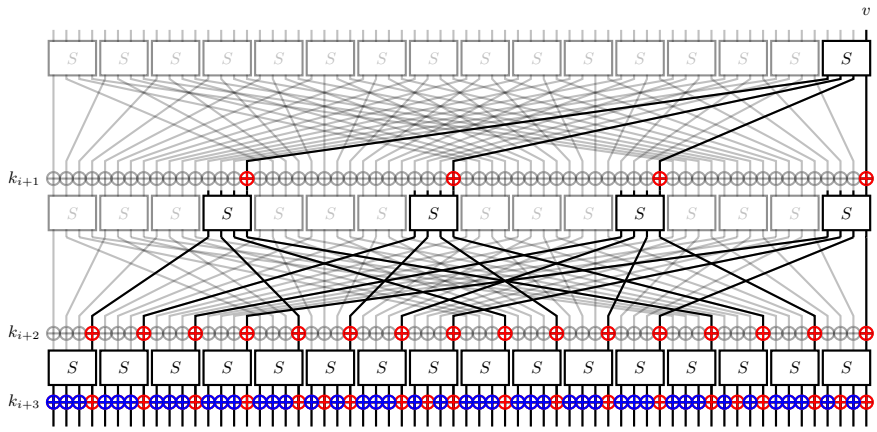


Fig. 7: 3-round key recovery on `PRESENT` using a divisibility property on the LSB. Red key bits denote bits that, due to being into α-admissible key differences, can be assumed not to be guessed, while blue bits are the ones that actually should remain to be guessed.

Best known attack. The best published integral key-recovery attack is due to Todo et al. [30]. Their attack targets 12 rounds using a 9-round integral distinguisher with a data complexity of 2^{60} and a mask of α=`0x1` (LSB), requiring the guessing of 82 key bits. The 82 bits are split via the *Match-through-the-S-box* technique (obtained by analyzing the ANF of the relevant S-box coordinate) – into two groups of 42 and 40 bits, which are handled in separate FFT evaluations of the parity. The overall cost becomes $3\times32\times2^{2+8+32} + 3\times32\times2^{8+32} \approx 2^{48.9}$ additions.

Running Algorithm 1 on this setting, we see that the α-admissible key difference subspace has dimension 150. In other words, out of the 192 bits involved in 3 round keys, *only 42* bits(0+0+42 across rounds, see Fig. 7) need to be guessed – almost twice less than one could expect. Surprisingly, no bits of the subkeys k_{10} and k_{11} need to be guessed. This reduction immediately improves the complexity of the key-recovery attack. After MTTS splitting into groups of 20 and 22 bits, the same key recovery steps now run in time of $3 \times 22 \times 2^{0+0+22} + 3 \times 20 \times 2^{0+0+20} = 2^{28.3}$ additions, instead of $2^{48.9}$.

We also considered a more advanced distinguisher of 2^{63} data complexity exploiting divisibility properties recently described by Beyne [7]. In particular, the rightmost four bits have divisibility properties with values v – $(2, 2, 2, 3)$ (where LSB's $v = 3$), leading to a stronger filter of 2^{-5}. In this setting, the naive analysis suggests guessing 84 key bits altogether $(4 + 16 + 64)$ as the MTTS approach cannot be applied anymore – because we check divisibility simultaneously at four bits. For the same reason, we compute the admissible subspace as the intersection of the per-bit subspaces, $R = R_{\alpha=\texttt{0x1}} \cap R_{\alpha=\texttt{0x2}} \cap R_{\alpha=\texttt{0x4}} \cap R_{\alpha=\texttt{0x8}}$, where each R_α is the α-admissible subspace returned by

Algorithm 1. This gives R of dimension 148, indicating that only $192 - 148 = 44$ bits $(0 + 0 + 44)$ need to be guessed. Consequently, the key-recovery phase can again be substantially accelerated: the time complexity drops from $2^{93.6}$ to 2^{53} (using FFT in both cases).

5.2 Application to GIFT

GIFT is a lightweight SPN cipher (4-bit S-box layer followed by a bit permutation). Although its natural description is bit-sliced, it can be drawn in the same round-function style as PRESENT (Fig. 8). A key structural difference is that the permutation wires keep the bit position within a nibble (MSB . . . LSB) aligned: bit $i \in \{0, 1, 2, 3\}$ of an S-box output always feeds bit i of some input nibble in the next round. The inverse GIFT S-box exhibits linear structures on the two most significant bits of a nibble.

Despite GIFT adding the round key only to half of the state, Algorithm 1 applies without modification. The only small change is at the end: the computed space is intersected with the key-injection subspace, i.e., the span of unit vectors at the bit positions where adding the key actually occurs in each round. This yields the space of α-admissible differences.

The design paper reports an integral key-recovery for 14 rounds using a 9-round distinguisher(which is automatically extendable to 10 rounds) on mask α=0x1 (LSB), which is still best-known. Although a 10-round distinguisher is given in [33], without a key-recovery phase, an identical attack can be executed for 15 rounds. For a fair comparison, we use the same partial sums technique as in the design paper for the key-recovery step.

For 14-round key-recovery (with 9-round distinguisher) the ARKST analysis yields two cross-round α-admissible key differences of weight 2, linking round 11 and round 12: a bit in k_{11} paired with a corresponding bit in k_{12} (see Fig. 8). Complementing both bits in a pair leaves all divisibility properties at the LSB unchanged, so only one bit per pair needs to be guessed. Consequently, the key-guessing complexity drops from 2^{96} to 2^{94} encryptions. For 15-round key-recovery (with 9-round distinguisher), results are identical, but with subkeys k_{12} and k_{13}.

5.3 Application to RECTANGLE

RECTANGLE is a lightweight block cipher introduced by Zhang et al. in 2015 [35]. It is designed as a bit-sliced cipher, but can also be represented in a similar round-function diagram as PRESENT (see Fig. 9). The main structural difference lies in the bit permutation: in RECTANGLE, each output bit of an S-box nibble connects to the same position with input in the next round nibble. Full diffusion is achieved only after four rounds, compared to 3 rounds in PRESENT. A notable property is that the inverse of the RECTANGLE S-box has linear structures on its 2nd and 3rd LSBs.

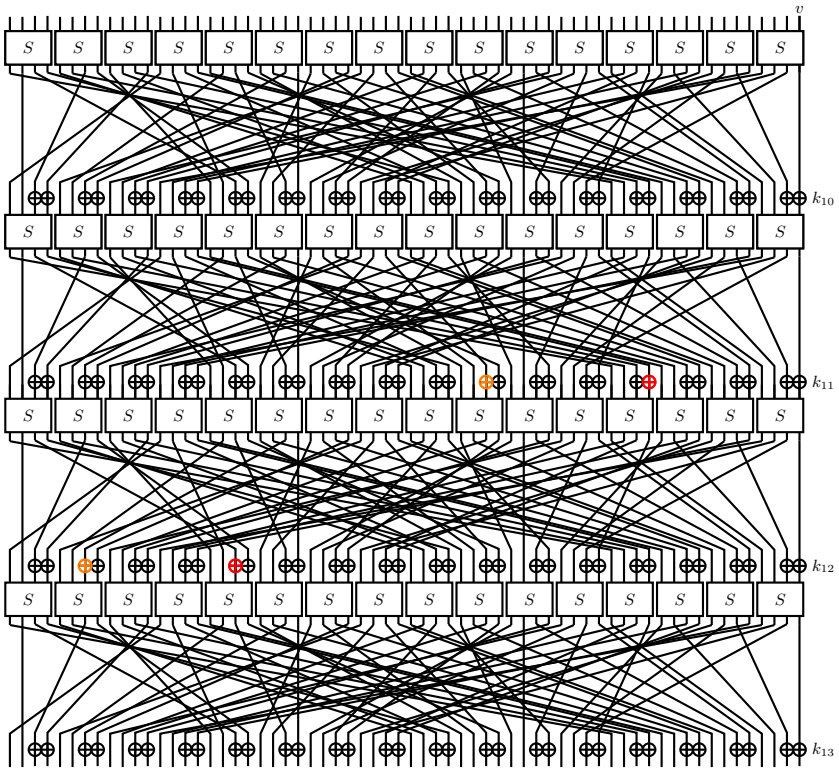


Fig. 8: 4-round key recovery on `GIFT` [2]. Two key bit pairs (red and orange, 4 bits total): complementing both bits in a pair (w.r.t. the correct key) preserves the integral property.

So far, no key-recovery attack has been published using the longest known 10-round integral distinguisher [24,33]; existing works only cover 8-round distinguishers [22]. To enable a fair comparison, we consider extending the basic FFT-based key-recovery (with mask α=`0x100`) to 13 and 14 rounds, with and without exploiting ARKST. For a 13-round variant, we recover the last 3 rounds (11-13); for a 14-round variant, we recover the last 4 rounds (11–14).

13-round attack. Without ARKST, the naive analysis requires guessing 56 key bits ($4 + 16 + 36$, across rounds). By applying ARKST, the number of involved key bits reduces to 48 ($2 + 14 + 32$). This directly lowers the FFT key-recovery complexity from $3 \times 36 \times 2^{56} \approx 2^{62.75}$ to $3 \times 32 \times 2^{48} \approx 2^{54.6}$.

14-round attack. Similarly, for 14 rounds, the naive key guessing requires 120 bits ($4 + 16 + 36 + 64$). Actually, this can be reduced to 108 bits ($2 + 14 + 32 + 60$). Again, the time complexity improves accordingly, from $3 \times 64 \times 2^{120} \approx 2^{127.6}$ to $3 \times 60 \times 2^{108} \approx 2^{115.5}$.

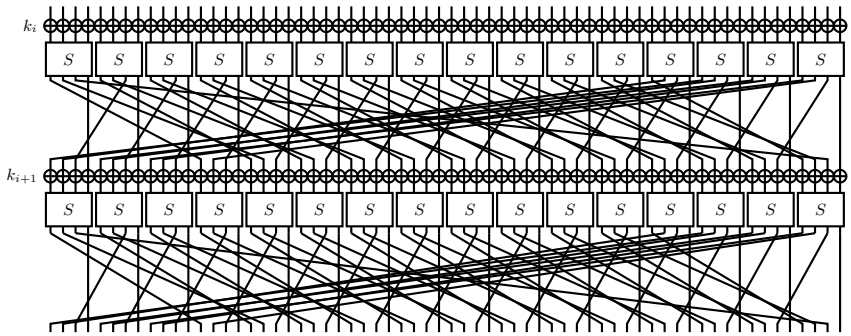


Fig. 9: RECTANGLE round function [35]

Use of generative AI. During the preparation of this article, OpenAI's *ChatGPT* (GPT-4-turbo and GPT-5, accessed in 2025, both free and paid versions) was used as a tool to support the research and writing process. More precisely, ChatGPT was employed for the following tasks: (1) basic text editing and language refinement, (2) assistance in exploring and deriving the proofs of the theorems presented herein, (3) drafting a preliminary outline of the theoretical framework for ARKST, (4) implementing Algorithm 1 from its description in Section 4. All conceptual development, verification of results, and final revisions were conducted by the authors, who take full responsibility of the content.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. Some figures were based on the templates in *TikZ for Cryptographers* [19].

This work was (in part) supported by the European Research Council (ERC) project 101097056 (SYMTRUST).

References

1. Ashur, T., Beyne, T., Rijmen, V.: Revisiting the wrong-key-randomization hypothesis. J. Cryptol. 33(2), 567–594 (2020), https://doi.org/10.1007/s00145-020-09343-2
2. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A small present. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017. LNCS, vol. 10529, pp. 321–345. Springer, Cham (2017), https://doi.org/10.1007/978-3-319-66787-4_16
3. Bar-On, A., Dunkelman, O., Keller, N., Weizman, A.: DLCT: A new tool for differential-linear cryptanalysis. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2019. LNCS, vol. 11476, pp. 313–342. Springer, Cham (2019), https://doi.org/10.1007/978-3-030-17653-2_11
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Paper 2013/404 (2013), https://eprint.iacr.org/2013/404

5. Beierle, C., Hebborn, P., Leander, G., Perehuda, Y.: Integral resistance of block ciphers with key whitening by modular addition. In: Kalai, Y.T., Kamara, S.F. (eds.) Advances in Cryptology - CRYPTO 2025. LNCS, vol. 16004, pp. 497–529. Springer, Cham (2025), https://doi.org/10.1007/978-3-032-01901-1_16
6. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology - CRYPTO 2016. LNCS, vol. 9815, pp. 123–153. Springer, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-53008-5_5
7. Beyne, T., Verbauwhede, M.: Ultrametric integral cryptanalysis. In: Chung, K., Sasaki, Y. (eds.) Advances in Cryptology - ASIACRYPT 2024. LNCS, vol. 15490, pp. 392–423. Springer, Singapore (2024), https://doi.org/10.1007/978-981-96-0941-3_13
8. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Berlin, Heidelberg (2007), https://doi.org/10.1007/978-3-540-74735-2_31
9. Bogdanov, A., Tischhauser, E.: On the wrong key randomisation and key equivalence hypotheses in Matsui's algorithm 2. In: Moriai, S. (ed.) Fast Software Encryption. FSE 2013. LNCS, vol. 8424, pp. 19–38. Springer, Berlin, Heidelberg (2014), https://doi.org/10.1007/978-3-662-43933-3_2
10. Boura, C., David, N., Derbez, P., Boissier, R.H., Naya-Plasencia, M.: A generic algorithm for efficient key recovery in differential attacks - and its associated tool. In: Joye, M., Leander, G. (eds.) Advances in Cryptology - EUROCRYPT 2024. LNCS, vol. 14651, pp. 217–248. Springer, Cham (2024), https://doi.org/10.1007/978-3-031-58716-0_8
11. Dunkelman, O., Ghosh, S., Keller, N., Leurent, G., Marmor, A., Mollimard, V.: Partial sums meet FFT: improved attack on 6-round AES. In: Joye, M., Leander, G. (eds.) Advances in Cryptology - EUROCRYPT 2024. LNCS, vol. 14651, pp. 128–157. Springer, Cham (2024), https://doi.org/10.1007/978-3-031-58716-0_5
12. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D.A., Whiting, D.: Improved cryptanalysis of Rijndael. In: Schneier, B. (ed.) Fast Software Encryption. FSE 2000. LNCS, vol. 1978, pp. 213–230. Springer, Berlin, Heidelberg (2001), https://doi.org/10.1007/3-540-44706-7_15
13. Gould, H.W.: Combinatorial identities: Table I: Intermediate techniques for summing finite series (May 2010), unpublished manuscripts compiled and edited by J. Quaintance. Available at https://math.wvu.edu/~hgould/Vol.4.PDF
14. Grassi, L., Rechberger, C., Rønjom, S.: Subspace trail cryptanalysis and its applications to AES. IACR Trans. Symmetric Cryptol. 2016(2), 192–225 (2016), https://doi.org/10.13154/tosc.v2016.i2.192-225
15. Grinstead, C.M., Snell, J.L.: Introduction to probability. American Mathematical Society (2012)
16. Guo, J., Liu, M., Song, L.: Linear structures: Applications to cryptanalysis of round-reduced Keccak. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016. LNCS, vol. 10031, pp. 249–274. Springer, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-53887-6_9
17. Hebborn, P., Lambin, B., Leander, G., Todo, Y.: Strong and tight security guarantees against integral distinguishers. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2021. LNCS, vol. 13090, pp. 362–391. Springer, Cham (2021), https://doi.org/10.1007/978-3-030-92062-3_13

18. Hu, K., Sun, S., Todo, Y., Wang, M., Wang, Q.: Massive superpoly recovery with nested monomial predictions. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2021. LNCS, vol. 13090, pp. 392–421. Springer, Cham (2021), https://doi.org/10.1007/978-3-030-92062-3_14
19. Jean, J.: TikZ for Cryptographers. https://www.iacr.org/authors/tikz/ (2016)
20. Knudsen, L.R.: Truncated and higher order differentials. In: Preneel, B. (ed.) Fast Software Encryption. FSE 1994. LNCS, vol. 1008, pp. 196–211. Springer, Berlin, Heidelberg (1995), https://doi.org/10.1007/3-540-60590-8_16
21. Knudsen, L.R., Wagner, D.A.: Integral cryptanalysis. In: Daemen, J., Rijmen, V. (eds.) Fast Software Encryption. FSE 2002. LNCS, vol. 2365, pp. 112–127. Springer, Berlin, Heidelberg (2002), https://doi.org/10.1007/3-540-45661-9_9
22. Kosuge, H., Tanaka, H., Iwai, K., Kurokawa, T.: Integral attack on reduced-round Rectangle. In: IEEE 2nd International Conference on Cyber Security and Cloud Computing, CSCloud 2015, New York, NY, USA, November 3-5, 2015. pp. 68–73. IEEE Computer Society (2015), https://doi.org/10.1109/CSCloud.2015.15
23. Lai, X.: Higher order derivatives and differential cryptanalysis. In: Blahut, R.E., Costello, D.J., Maurer, U., Mittelholzer, T. (eds.) Communications and Cryptography: Two Sides of One Tapestry, pp. 227–233. Springer US, Boston, MA (1994), https://doi.org/10.1007/978-1-4615-2694-0_23
24. Lambin, B., Derbez, P., Fouque, P.: Linearly equivalent s-boxes and the division property. Des. Codes Cryptogr. 88(10), 2207–2231 (2020), https://doi.org/10.1007/s10623-020-00773-4
25. Lambooij, E., Neumann, P., Verbauwhede, M., Wang, S., Zhang, T.: Meet-in-the-middle attacks on full ChiLow. Cryptology ePrint Archive, Paper 2025/1601 (2025), https://eprint.iacr.org/2025/1601
26. Leander, G., Tezcan, C., Wiemer, F.: Searching for subspace trails and truncated differentials. IACR Trans. Symmetric Cryptol. 2018(1), 74–100 (2018), https://doi.org/10.13154/tosc.v2018.i1.74-100
27. Tezcan, C., Özbudak, F.: Differential factors: Improved attacks on SERPENT. In: Eisenbarth, T., Öztürk, E. (eds.) Lightweight Cryptography for Security and Privacy. LightSec 2014. LNCS, vol. 8898, pp. 69–84. Springer, Cham (2015), https://doi.org/10.1007/978-3-319-16363-5_5
28. Todo, Y.: Structural evaluation by generalized integral property. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015. LNCS, vol. 9056, pp. 287–314. Springer, Berlin, Heidelberg (2015), https://doi.org/10.1007/978-3-662-46800-5_12
29. Todo, Y., Aoki, K.: FFT key recovery for integral attack. In: Gritzalis, D., Kiayias, A., Askoxylakis, I.G. (eds.) Cryptology and Network Security. CANS 2014. LNCS, vol. 8813, pp. 64–81. Springer, Cham (2014), https://doi.org/10.1007/978-3-319-12280-9_5
30. Todo, Y., Morii, M.: Compact representation for division property. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security. CANS 2016. LNCS, vol. 10052, pp. 19–35. Springer, Cham (2016), https://doi.org/10.1007/978-3-319-48965-0_2
31. Wang, Q., Liu, Z., Varici, K., Sasaki, Y., Rijmen, V., Todo, Y.: Cryptanalysis of reduced-round SIMON32 and SIMON48. In: Meier, W., Mukhopadhyay, D. (eds.) Progress in Cryptology - INDOCRYPT 2014. LNCS, vol. 8885, pp. 143–160. Springer, Cham (2014), https://doi.org/10.1007/978-3-319-13039-2_9
32. Wilf, H.S.: generatingfunctionology. CRC press (2005)

33. Zhang, L., Liu, H., Wang, Z.: A model set method to search integral distinguishers based on division property for block ciphers. Cryptology ePrint Archive, Paper 2022/720 (2022), https://eprint.iacr.org/2022/720
34. Zhang, L., Yao, Y., Shi, D., Chai, D., Guo, J., Wang, Z.: Neural-inspired advances in integral cryptanalysis. Cryptology ePrint Archive, Paper 2025/852 (2025), https://eprint.iacr.org/2025/852
35. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. Sci. China Inf. Sci. 58(12), 1–15 (2015), https://doi.org/10.1007/s11432-015-5459-7

A Application to SIMON32

SIMON was publicly released by the U.S. NSA in 2013 [4] as part of the SIMON/SPECK family of lightweight block ciphers, aimed at area-constrained devices. The family offers multiple block/key sizes (blocks from 32 to 128 bits; keys from 64 to 256 bits). In this section, we analyse the SIMON32/64 instance. The design is a balanced Feistel cipher with a round function (Fig. 10) built solely from word rotations, bitwise AND, and XOR:

$$F(x) \;=\; (x \lll 1)\,\&\,(x \lll 8) \;\oplus\; (x \lll 2).$$

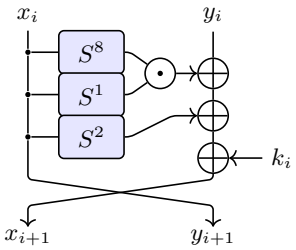


Fig. 10: SIMON round function [19]

In this subsection, we do not run ARKST. Instead, we reanalyse an ANF of the partial decryption of integral key-recovery. We follow the attack [31] which uses a 15-round distinguisher with 2^{31} chosen plaintexts yielding a zero-sum on the LSB, i.e. $\bigoplus R_{15,\{0\}} = 0$. A naive last-round key recovery would guess 49 bits, but the paper applies a meet-in-the-middle split on the parity

$$\bigoplus L_{15,\{15\}} \;\&\; L_{15,\{8\}} \;=\; \bigoplus L_{15,\{14\}} \;\oplus\; L_{16,\{0\}},$$

and handles the two sides separately with partial sums. Each side is claimed to depend on 42 key bits.

Optimisation of key-guessing part. To compute the XOR side $L_{15,\{14\}} \oplus L_{16,\{0\}}$ one must also know $k_{18,\{11\}}, k_{19,\{9\}}, k_{20,\{7\}}$; hence this side actually depends on 45 bits, not 42, which was not covered in [31]. Under the cost model of [31], this lifts the original key-recovery time to about $2^{54.6}$ 21-round encryptions.

Write the partial decryption for the XOR side in ANF and propagate a few rounds:

$$\bigoplus L_{15,\{14\}} \oplus L_{16,\{0\}} = \bigoplus\big(\ldots \oplus k_{16,\{14\}} \oplus k_{17,\{0\}} \oplus k_{17,\{12\}} \oplus k_{18,\{10\}} \oplus k_{19,\{8\}}\big)$$

The five listed key bits appear *linearly* and do not occur inside any nonlinear monomials. Flipping any of them toggles both sides of the equation equally,

leaving the parity test unchanged. Hence, they are "neutral" and need not be guessed, the XOR side reduces from 45 to 37 effective bits.

For the AND side, one derives

$$\begin{aligned}\bigoplus L_{15,\{15\}} \;\&\; L_{15,\{8\}} &= \bigoplus R_{16,\{15\}} \;\&\; R_{16,\{8\}} \\ &= \bigoplus \Big(\cdots \oplus \underbrace{k_{16,\{15\}} \oplus k_{17,\{13\}} \oplus k_{18,\{11\}} \oplus k_{19,\{9\}} \oplus k_{20,\{7\}}}_{\Sigma_{15}}\Big) \\ &\quad \;\&\; \Big(\cdots \oplus \underbrace{k_{16,\{8\}} \oplus k_{17,\{6\}} \oplus k_{18,\{4\}} \oplus k_{19,\{2\}} \oplus k_{20,\{0\}}}_{\Sigma_8}\Big).\end{aligned}$$

One can reduce each of these 5-bit sums to guessing just one key bit, regardless of the remaining terms. Thus, each set defines a 4-dimensional α-admissible key difference subspace; from each group of five, it suffices to guess one bit (preferably the closest round to the zero-sum property for stronger complexity effect). Consequently, the AND side also drops from 45 to 37 effective bits.

Now we recalculate partial-sum complexities of [31] with taking into account the smaller guessed key subspace. AND side yields the following step costs:

$$\begin{aligned}&\text{Step 1:} \quad 2^{40.51} \quad (\text{was } 2^{42.51}) \\ &\text{Step 2:} \quad 2^{47.19} \quad (\text{was } 2^{51.19}) \\ &\text{Step 3:} \quad 2^{46.61} \quad (\text{was } 2^{52.61}) \\ &\text{Step 4:} \quad 2^{41.93} \quad (\text{was } 2^{49.93}) \\ &\text{Step 5:} \quad 2^{37.19} \quad (\text{was } 2^{45.19})\end{aligned}$$

so this branch has time complexity about $2^{47.96}$.

For the XOR side resulting costs are

$$\begin{aligned}&\text{Step 1:} \quad 2^{42.61} \quad (\text{was } 2^{42.61}) \\ &\text{Step 2:} \quad 2^{52.31} \quad (\text{was } 2^{53.31}) \\ &\text{Step 3:} \quad 2^{50.61} \quad (\text{was } 2^{52.61}) \\ &\text{Step 4:} \quad 2^{42.61} \quad (\text{was } 2^{46.61}) \\ &\text{Step 5:} \quad 2^{34.61} \quad (\text{was } 2^{39.61})\end{aligned}$$

so this branch is dominated by about $2^{51.31}$.

With these reductions, the key-recovery phase drops from roughly $2^{54.6}$ to $\mathbf{2^{52.75}}$ 21-round SIMON32 computations.

B Explicit bases of α-admissible key-difference subspaces for 3 rounds of key-recovery

Below we list a basis in echelon form for the subspace of α-admissible key differences in the first three key-recovery rounds. Each triple is $(\Delta k_1, \Delta k_2, \Delta k_3)$, written as 64-bit hex.

B.1 PRESENT

```
1  (0x8000000000000000, 0x0000000000000000, 0x0000000000000000)
2  (0x4000000000000000, 0x0000000000000000, 0x0000000000000000)
3  (0x2000000000000000, 0x0000000000000000, 0x0000000000000000)
4  (0x1000000000000000, 0x0000000000000000, 0x0000000000000000)
5  (0x0800000000000000, 0x0000000000000000, 0x0000000000000000)
6  (0x0400000000000000, 0x0000000000000000, 0x0000000000000000)
7  (0x0200000000000000, 0x0000000000000000, 0x0000000000000000)
8  (0x0100000000000000, 0x0000000000000000, 0x0000000000000000)
9  (0x0080000000000000, 0x0000000000000000, 0x0000000000000000)
10 (0x0040000000000000, 0x0000000000000000, 0x0000000000000000)
11 (0x0020000000000000, 0x0000000000000000, 0x0000000000000000)
12 (0x0010000000000000, 0x0000000000000000, 0x0000000000000000)
13 (0x0008000000000000, 0x0000000000000000, 0x0000000000000000)
14 (0x0004000000000000, 0x0000000000000000, 0x0000000000000000)
15 (0x0002000000000000, 0x0000000000000000, 0x0000000000000000)
16 (0x0001000000000000, 0x0000000000000000, 0x0000000000000008)
17 (0x0000800000000000, 0x0000000000000000, 0x0000000000000000)
18 (0x0000400000000000, 0x0000000000000000, 0x0000000000000000)
19 (0x0000200000000000, 0x0000000000000000, 0x0000000000000000)
20 (0x0000100000000000, 0x0000000000000000, 0x0000000000000000)
21 (0x0000080000000000, 0x0000000000000000, 0x0000000000000000)
22 (0x0000040000000000, 0x0000000000000000, 0x0000000000000000)
23 (0x0000020000000000, 0x0000000000000000, 0x0000000000000000)
24 (0x0000010000000000, 0x0000000000000000, 0x0000000000000000)
25 (0x0000008000000000, 0x0000000000000000, 0x0000000000000000)
26 (0x0000004000000000, 0x0000000000000000, 0x0000000000000000)
27 (0x0000002000000000, 0x0000000000000000, 0x0000000000000000)
28 (0x0000001000000000, 0x0000000000000000, 0x0000000000000000)
29 (0x0000000800000000, 0x0000000000000000, 0x0000000000000000)
30 (0x0000000400000000, 0x0000000000000000, 0x0000000000000000)
31 (0x0000000200000000, 0x0000000000000000, 0x0000000000000000)
32 (0x0000000100000000, 0x0000000000000000, 0x0000000000000000)
33 (0x0000000080000000, 0x0000000000000000, 0x0000000000000000)
34 (0x0000000040000000, 0x0000000000000000, 0x0000000000000000)
35 (0x0000000020000000, 0x0000000000000000, 0x0000000000000000)
36 (0x0000000010000000, 0x0000000000000000, 0x0000000000000000)
37 (0x0000000008000000, 0x0000000000000000, 0x0000000000000000)
38 (0x0000000004000000, 0x0000000000000000, 0x0000000000000000)
39 (0x0000000002000000, 0x0000000000000000, 0x0000000000000000)
```

```
(0x0000000001000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000800000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000400000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000200000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000100000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000080000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000040000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000020000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000010000, 0x0000000000000000, 0x0000000000000002)
(0x0000000000008000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000004000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000002000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000001000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000800, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000400, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000200, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000100, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000080, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000040, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000020, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000010, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000008, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000004, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000002, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000001, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000000, 0x8000000000000000, 0x0000000000000000)
(0x0000000000000000, 0x4000000000000000, 0x0000000000000000)
(0x0000000000000000, 0x2000000000000000, 0x0000000000000000)
(0x0000000000000000, 0x1000000000000000, 0x0000000000008000)
(0x0000000000000000, 0x0800000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0400000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0200000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0100000000000000, 0x0000000000004000)
(0x0000000000000000, 0x0080000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0040000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0020000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0010000000000000, 0x0000000000002000)
(0x0000000000000000, 0x0008000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0004000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0002000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0001000000000000, 0x0000000000001000)
(0x0000000000000000, 0x0000800000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000400000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000200000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000100000000000, 0x0000000000000008)
(0x0000000000000000, 0x0000080000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000040000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000020000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000010000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000008000000000, 0x0000000000000000)
```

```
90  (0x0000000000000000, 0x0000004000000000, 0x0000000000000000)
91  (0x0000000000000000, 0x0000002000000000, 0x0000000000000000)
92  (0x0000000000000000, 0x0000001000000000, 0x0000000000000002)
93  (0x0000000000000000, 0x0000000800000000, 0x0000000000000000)
94  (0x0000000000000000, 0x0000000400000000, 0x0000000000000000)
95  (0x0000000000000000, 0x0000000200000000, 0x0000000000000000)
96  (0x0000000000000000, 0x0000000100000000, 0x0000000000000000)
97  (0x0000000000000000, 0x0000000080000000, 0x0000000000000000)
98  (0x0000000000000000, 0x0000000040000000, 0x0000000000000000)
99  (0x0000000000000000, 0x0000000020000000, 0x0000000000000000)
100 (0x0000000000000000, 0x0000000010000000, 0x0000000000000080)
101 (0x0000000000000000, 0x0000000008000000, 0x0000000000000000)
102 (0x0000000000000000, 0x0000000004000000, 0x0000000000000000)
103 (0x0000000000000000, 0x0000000002000000, 0x0000000000000000)
104 (0x0000000000000000, 0x0000000001000000, 0x0000000000000040)
105 (0x0000000000000000, 0x0000000000800000, 0x0000000000000000)
106 (0x0000000000000000, 0x0000000000400000, 0x0000000000000000)
107 (0x0000000000000000, 0x0000000000200000, 0x0000000000000000)
108 (0x0000000000000000, 0x0000000000100000, 0x0000000000000020)
109 (0x0000000000000000, 0x0000000000080000, 0x0000000000000000)
110 (0x0000000000000000, 0x0000000000040000, 0x0000000000000000)
111 (0x0000000000000000, 0x0000000000020000, 0x0000000000000000)
112 (0x0000000000000000, 0x0000000000010000, 0x0000000000000010)
113 (0x0000000000000000, 0x0000000000008000, 0x0000000000000000)
114 (0x0000000000000000, 0x0000000000004000, 0x0000000000000000)
115 (0x0000000000000000, 0x0000000000002000, 0x0000000000000000)
116 (0x0000000000000000, 0x0000000000001000, 0x0000000000000008)
117 (0x0000000000000000, 0x0000000000000800, 0x0000000000000000)
118 (0x0000000000000000, 0x0000000000000400, 0x0000000000000000)
119 (0x0000000000000000, 0x0000000000000200, 0x0000000000000000)
120 (0x0000000000000000, 0x0000000000000100, 0x0000000000000000)
121 (0x0000000000000000, 0x0000000000000080, 0x0000000000000000)
122 (0x0000000000000000, 0x0000000000000040, 0x0000000000000000)
123 (0x0000000000000000, 0x0000000000000020, 0x0000000000000000)
124 (0x0000000000000000, 0x0000000000000010, 0x0000000000000002)
125 (0x0000000000000000, 0x0000000000000008, 0x0000000000000000)
126 (0x0000000000000000, 0x0000000000000004, 0x0000000000000000)
127 (0x0000000000000000, 0x0000000000000002, 0x0000000000000000)
128 (0x0000000000000000, 0x0000000000000001, 0x0000000000000000)
129 (0x0000000000000000, 0x0000000000000000, 0x0000800000008000)
130 (0x0000000000000000, 0x0000000000000000, 0x0000400000004000)
131 (0x0000000000000000, 0x0000000000000000, 0x0000200000002000)
132 (0x0000000000000000, 0x0000000000000000, 0x0000100000001000)
133 (0x0000000000000000, 0x0000000000000000, 0x0000080000000008)
134 (0x0000000000000000, 0x0000000000000000, 0x0000040000000000)
135 (0x0000000000000000, 0x0000000000000000, 0x0000020000000002)
136 (0x0000000000000000, 0x0000000000000000, 0x0000010000000000)
137 (0x0000000000000000, 0x0000000000000000, 0x0000008000000080)
138 (0x0000000000000000, 0x0000000000000000, 0x0000004000000040)
139 (0x0000000000000000, 0x0000000000000000, 0x0000002000000020)
```

```
(0x0000000000000000, 0x0000000000000000, 0x0000001000000010)
(0x0000000000000000, 0x0000000000000000, 0x0000000800000008)
(0x0000000000000000, 0x0000000000000000, 0x0000000400000000)
(0x0000000000000000, 0x0000000000000000, 0x0000000200000002)
(0x0000000000000000, 0x0000000000000000, 0x0000000100000000)
(0x0000000000000000, 0x0000000000000000, 0x0000000000000808)
(0x0000000000000000, 0x0000000000000000, 0x0000000000000400)
(0x0000000000000000, 0x0000000000000000, 0x0000000000000202)
(0x0000000000000000, 0x0000000000000000, 0x0000000000000100)
(0x0000000000000000, 0x0000000000000000, 0x0000000000000004)
(0x0000000000000000, 0x0000000000000000, 0x0000000000000001)
```

B.2 GIFT

```
(0x2000000000000000, 0x0000000000000000, 0x0000000000000000)
(0x1000000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0200000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0100000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0020000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0010000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0002000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0001000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000200000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000100000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000020000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000010000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000002000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000001000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000200000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000100000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000020000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000010000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000002000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000001000000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000200000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000100000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000020000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000010000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000002000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000001000, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000200, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000100, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000020, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000010, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000001, 0x0000000000000000, 0x0000000000000000)
(0x0000000000000000, 0x2000000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0200000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0100000000000000, 0x0000002000000000)
(0x0000000000000000, 0x0020000000000000, 0x0000000000000000)
```

```
(0x0000000000000000, 0x0002000000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000200000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000100000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000020000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000010000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000002000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000001000000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000200000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000100000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000020000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000010000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000002000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000001000000, 0x0000000000000000)
(0x0000000000000000, 0x0000000000200000, 0x0000000000000000)
(0x0000000000000000, 0x0000000000100000, 0x0000000000000000)
(0x0000000000000000, 0x0000000000020000, 0x0000000000000000)
(0x0000000000000000, 0x0000000000010000, 0x0000000000000000)
(0x0000000000000000, 0x0000000000001000, 0x0000000000000000)
(0x0000000000000000, 0x0000000000000100, 0x0000000000000000)
(0x0000000000000000, 0x0000000000000020, 0x0020000000000000)
(0x0000000000000000, 0x0000000000000010, 0x0000000000000000)
(0x0000000000000000, 0x0000000000000001, 0x0000000000000000)
```

B.3 RECTANGLE

```
(0x8000000000000000, 0x0000000000000000, 0x0000000000000000)
(0x4000000000000000, 0x0000000000000000, 0x0000000000000000)
(0x2000000000000000, 0x0000000000000000, 0x0000000000000000)
(0x1000000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0800000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0400000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0200000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0100000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0080000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0040000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0020000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0010000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0008000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0004000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0002000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0001000000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000800000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000400000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000200000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000100000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000080000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000040000000000, 0x0000000000000000, 0x0000000000000000)
(0x0000020000000000, 0x0000000000000000, 0x0000000000000000)
```

```
24 (0x0000010000000000, 0x0000000000000000, 0x0000000000000000)
25 (0x0000008000000000, 0x0000000000000000, 0x0000000000000000)
26 (0x0000004000000000, 0x0000000000000000, 0x0000000000000000)
27 (0x0000002000000000, 0x0000000000000000, 0x0000000000000000)
28 (0x0000001000000000, 0x0000000000000000, 0x0000000000000000)
29 (0x0000000800000000, 0x0000000000000000, 0x0000000000000000)
30 (0x0000000400000000, 0x0000000000000000, 0x0000000000000000)
31 (0x0000000200000000, 0x0000000000000000, 0x0000000000000000)
32 (0x0000000100000000, 0x0000000000000000, 0x0000000000000000)
33 (0x0000000080000000, 0x0000000000000000, 0x0000000000000000)
34 (0x0000000040000000, 0x0000000000000000, 0x0000000000000000)
35 (0x0000000020000000, 0x0000000000000000, 0x0000000000000000)
36 (0x0000000010000000, 0x0000000000000000, 0x0000000000000000)
37 (0x0000000008000000, 0x0000000000000000, 0x0000000000000000)
38 (0x0000000004000000, 0x0000000000000000, 0x0000000000000000)
39 (0x0000000002000000, 0x0000000000000000, 0x0000084000000000)
40 (0x0000000001000000, 0x0000000000000000, 0x0000000000000000)
41 (0x0000000000800000, 0x0000000000000000, 0x0000000000000000)
42 (0x0000000000400000, 0x0000000000000000, 0x0000000000000000)
43 (0x0000000000200000, 0x0000000000000000, 0x0000000000000000)
44 (0x0000000000080000, 0x0000000000000000, 0x0000000000000000)
45 (0x0000000000040000, 0x0000000000000000, 0x0000000000000000)
46 (0x0000000000020000, 0x0000000000000000, 0x0000000000000000)
47 (0x0000000000010000, 0x0000000000000000, 0x0000000000000000)
48 (0x0000000000008000, 0x0000000000000000, 0x0000000000000000)
49 (0x0000000000004000, 0x0000000000000000, 0x0000000000000000)
50 (0x0000000000002000, 0x0000000000000000, 0x0000000000000000)
51 (0x0000000000001000, 0x0000000000000000, 0x0000000000000000)
52 (0x0000000000000400, 0x0000000000000000, 0x0000000000000000)
53 (0x0000000000000200, 0x0000000000000000, 0x0000000000000000)
54 (0x0000000000000100, 0x0000000000000000, 0x0000000000000000)
55 (0x0000000000000080, 0x0000000000000000, 0x0000000000000000)
56 (0x0000000000000040, 0x0000000000000000, 0x0000000000020000)
57 (0x0000000000000020, 0x0000000000000000, 0x0000000000000000)
58 (0x0000000000000010, 0x0000000000000000, 0x0000000000000000)
59 (0x0000000000000008, 0x0000000000000000, 0x0000000000000000)
60 (0x0000000000000004, 0x0000000000000000, 0x0000000000000000)
61 (0x0000000000000002, 0x0000000000000000, 0x0000000000000000)
62 (0x0000000000000001, 0x0000000000000000, 0x0000000000000000)
63 (0x0000000000000000, 0x8000000000000000, 0x0000000000000000)
64 (0x0000000000000000, 0x4000000000000000, 0x0000000000000000)
65 (0x0000000000000000, 0x2000000000000000, 0x0000000000000000)
66 (0x0000000000000000, 0x1000000000000000, 0x0000000000000000)
67 (0x0000000000000000, 0x0800000000000000, 0x0000000000000000)
68 (0x0000000000000000, 0x0400000000000000, 0x0000000000000000)
69 (0x0000000000000000, 0x0200000000000000, 0x0000000000000000)
70 (0x0000000000000000, 0x0100000000000000, 0x0000000000000000)
71 (0x0000000000000000, 0x0080000000000000, 0x0000000000000000)
72 (0x0000000000000000, 0x0040000000000000, 0x0000000000000000)
73 (0x0000000000000000, 0x0020000000000000, 0x0000000000000000)
```

```
74 (0x0000000000000000, 0x0010000000000000, 0x0000000000000000)
75 (0x0000000000000000, 0x0008000000000000, 0x0000000000000000)
76 (0x0000000000000000, 0x0004000000000000, 0x0000000000000000)
77 (0x0000000000000000, 0x0002000000000000, 0x0000000000000000)
78 (0x0000000000000000, 0x0001000000000000, 0x0000000000000000)
79 (0x0000000000000000, 0x0000800000000000, 0x0000000000000000)
80 (0x0000000000000000, 0x0000400000000000, 0x0000000000000000)
81 (0x0000000000000000, 0x0000200000000000, 0x0000000000000000)
82 (0x0000000000000000, 0x0000100000000000, 0x0000000000000000)
83 (0x0000000000000000, 0x0000080000000000, 0x0000000000000000)
84 (0x0000000000000000, 0x0000040000000000, 0x0000000000000000)
85 (0x0000000000000000, 0x0000020000000000, 0x0000084000000000)
86 (0x0000000000000000, 0x0000010000000000, 0x0000000000000000)
87 (0x0000000000000000, 0x0000008000000000, 0x0000000000000000)
88 (0x0000000000000000, 0x0000004000000000, 0x0000000000000000)
89 (0x0000000000000000, 0x0000000800000000, 0x0000000000000000)
90 (0x0000000000000000, 0x0000000400000000, 0x0000000000000000)
91 (0x0000000000000000, 0x0000000200000000, 0x0000000000000000)
92 (0x0000000000000000, 0x0000000080000000, 0x0000000000000000)
93 (0x0000000000000000, 0x0000000040000000, 0x0000000000000000)
94 (0x0000000000000000, 0x0000000020000000, 0x0000000000000000)
95 (0x0000000000000000, 0x0000000010000000, 0x0000000000000000)
96 (0x0000000000000000, 0x0000000008400000, 0x0000084000000000)
97 (0x0000000000000000, 0x0000000004000000, 0x0000000000000000)
98 (0x0000000000000000, 0x0000000001000000, 0x0000000000000000)
99 (0x0000000000000000, 0x0000000000200000, 0x0000000000020000)
100 (0x0000000000000000, 0x0000000000080000, 0x0000000000000000)
101 (0x0000000000000000, 0x0000000000020000, 0x0000000000000000)
102 (0x0000000000000000, 0x0000000000008000, 0x0000000000000000)
103 (0x0000000000000000, 0x0000000000004000, 0x0000000000000000)
104 (0x0000000000000000, 0x0000000000002000, 0x0000000000000000)
105 (0x0000000000000000, 0x0000000000001000, 0x0000000000000000)
106 (0x0000000000000000, 0x0000000000000400, 0x0000000000000000)
107 (0x0000000000000000, 0x0000000000000200, 0x0000000000000000)
108 (0x0000000000000000, 0x0000000000000100, 0x0000000000000000)
109 (0x0000000000000000, 0x0000000000000020, 0x0000000000000000)
110 (0x0000000000000000, 0x0000000000000010, 0x0000000000000000)
111 (0x0000000000000000, 0x0000000000000008, 0x0000000000000000)
112 (0x0000000000000000, 0x0000000000000004, 0x0000000000020000)
113 (0x0000000000000000, 0x0000000000000002, 0x0000000000000000)
114 (0x0000000000000000, 0x0000000000000001, 0x0000000000000000)
115 (0x0000000000000000, 0x0000000000000000, 0x8000000000000000)
116 (0x0000000000000000, 0x0000000000000000, 0x4000000000020000)
117 (0x0000000000000000, 0x0000000000000000, 0x2000000000000000)
118 (0x0000000000000000, 0x0000000000000000, 0x1000000000000000)
119 (0x0000000000000000, 0x0000000000000000, 0x0800000000000000)
120 (0x0000000000000000, 0x0000000000000000, 0x0400000000000000)
121 (0x0000000000000000, 0x0000000000000000, 0x0200084000000000)
122 (0x0000000000000000, 0x0000000000000000, 0x0100000000000000)
123 (0x0000000000000000, 0x0000000000000000, 0x0080000000000000)
```

```
124 (0x0000000000000000, 0x0000000000000000, 0x0040000000000000)
125 (0x0000000000000000, 0x0000000000000000, 0x0008000000000000)
126 (0x0000000000000000, 0x0000000000000000, 0x0004000000000000)
127 (0x0000000000000000, 0x0000000000000000, 0x0000800000000000)
128 (0x0000000000000000, 0x0000000000000000, 0x0000400000000000)
129 (0x0000000000000000, 0x0000000000000000, 0x0000200000000000)
130 (0x0000000000000000, 0x0000000000000000, 0x0000040000000000)
131 (0x0000000000000000, 0x0000000000000000, 0x0000010000000000)
132 (0x0000000000000000, 0x0000000000000000, 0x0000000080000000)
133 (0x0000000000000000, 0x0000000000000000, 0x0000000020000000)
134 (0x0000000000000000, 0x0000000000000000, 0x0000000004000000)
135 (0x0000000000000000, 0x0000000000000000, 0x0000000001000000)
136 (0x0000000000000000, 0x0000000000000000, 0x0000000000008000)
137 (0x0000000000000000, 0x0000000000000000, 0x0000000000002000)
138 (0x0000000000000000, 0x0000000000000000, 0x0000000000000400)
139 (0x0000000000000000, 0x0000000000000000, 0x0000000000000200)
140 (0x0000000000000000, 0x0000000000000000, 0x0000000000000100)
141 (0x0000000000000000, 0x0000000000000000, 0x0000000000000020)
142 (0x0000000000000000, 0x0000000000000000, 0x0000000000000010)
143 (0x0000000000000000, 0x0000000000000000, 0x0000000000000002)
144 (0x0000000000000000, 0x0000000000000000, 0x0000000000000001)
```