Cube Testers and Key Recovery Attacks On Reduced-Round MD6 and Trivium

Jean-Philippe Aumasson[1*], Itai Dinur[2], Willi Meier[1†], and Adi Shamir[2]

[1] FHNW, Windisch, Switzerland
[2] Computer Science Department, The Weizmann Institute, Rehovot, Israel

Abstract. CRYPTO 2008 saw the introduction of the hash function MD6 and of cube attacks, a type of algebraic attack applicable to cryptographic functions having a low-degree algebraic normal form over GF(2). This paper applies cube attacks to reduced round MD6, finding the full 128-bit key of a 14-round MD6 with complexity 2^{22} (which takes less than a minute on a single PC). This is the best key recovery attack announced so far for MD6. We then introduce a new class of attacks called cube testers, based on efficient property-testing algorithms, and apply them to MD6 and to the stream cipher Trivium. Unlike the standard cube attacks, cube testers detect nonrandom behavior rather than performing key extraction, but they can also attack cryptographic schemes described by nonrandom polynomials of relatively high degree. Applied to MD6, cube testers detect nonrandomness over 18 rounds in 2^{17} complexity; applied to a slightly modified version of the MD6 compression function, they can distinguish 66 rounds from random in 2^{24} complexity. Cube testers give distinguishers on Trivium reduced to 790 rounds from random with 2^{30} complexity and detect nonrandomness over 885 rounds in 2^{27}, improving on the original 767-round cube attack.

1 Introduction

1.1 Cube Attacks

Cube attacks [1, 2] are a new type of algebraic cryptanalysis that exploit implicit low-degree equations in cryptographic algorithms. Cube attacks only require black box access to the target primitive, and were successfully applied to reduced versions of the stream cipher Trivium [3] in [2]. Roughly speaking, a cryptographic function is vulnerable to cube attacks if its implicit algebraic normal form over GF(2) has degree at most d, provided that 2^d computations of the function is feasible. Cube attacks recover a secret key through queries to a *black box polynomial* with *tweakable public variables* (e.g. chosen plaintext or IV bits), followed by solving a linear system of equations in the secret key variables. A one time preprocessing phase is required to determine which queries should be made to the black box during the on-line phase of the attack. Low-degree

*Supported by the Swiss National Science Foundation, project no. 113329.
†Supported by GEBERT RÜF STIFTUNG, project no. GRS-069/07.

implicit equations were previously exploited in [4–7] to construct distinguishers, and in [8–10] for key recovery. Cube attacks are related to saturation attacks [11] and to high order differential cryptanalysis [12].

Basics. Let $\mathcal{F}_n$ be the set of all functions mapping $\{0,1\}^n$ to $\{0,1\}$, $n > 0$, and let $f \in \mathcal{F}_n$. The *algebraic normal form* (ANF) of f is the polynomial p over GF(2) in variables $x_1, \ldots, x_n$ such that evaluating p on $x \in \{0,1\}^n$ is equivalent to computing $f(x)$, and such that it is of the form[3]

$$\sum_{i=0}^{2^n-1} a_i \cdot x_1^{i_1} x_2^{i_2} \cdots x_{n-1}^{i_{n-1}} x_n^{i_n}$$

for some $(a_0, \ldots, a_{2^n-1}) \in \{0,1\}^{2^n}$, and where i_j denotes the j-th digit of the binary encoding of i (and so the sum spans all monomials in $x_1, \ldots, x_n$). A key observation regarding cube attacks is that for any function $f : \{0,1\}^n \mapsto \{0,1\}$, the sum (XOR) of all entries in the truth table

$$\sum_{x \in \{0,1\}^n} f(x)$$

equals the coefficient of the highest degree monomial $x_1 \cdots x_n$ in the algebraic normal form (ANF) of f. For example, let $n = 4$ and f be defined as

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3 \ .$$

Then summing $f(x_1, x_2, x_3, x_4)$ over all 16 distinct inputs makes all monomials vanish and yields zero, i.e. the coefficient of the monomial $x_1x_2x_3x_4$. Instead, cube attacks sum over a *subset* of the inputs; for example summing over the four possible values of (x_1, x_2) gives

$$f(0,0,x_3,x_4) + f(0,1,x_3,x_4) + f(1,0,x_3,x_4) + f(1,1,x_3,x_4) = x_3 + x_4 \ ,$$

where $(x_3 + x_4)$ is the polynomial that multiplies x_1x_2 in f:

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2(x_3 + x_4) + x_3 \ .$$

Generalizing, given an index set $I \subsetneq \{1, \ldots, n\}$, any function in $\mathcal{F}_n$ can be represented algebraically under the form

$$f(x_1, \ldots, x_n) = t_I \cdot p(\cdots) + q(x_1, \ldots, x_n)$$

where t_I is the monomial containing all the x_i's with $i \in I$, p is a polynomial that has no variable in common with t_I, and such that no monomial in the

[3]The ANF of any $f \in \mathcal{F}_n$ has degree at most n, since $x_i^d = x_i$, for $x_i \in \mathrm{GF}(2)$, $d > 0$.

polynomial q contains t_I (that is, we factored f by the monomial t_I). Summing f over the *cube* t_I for other variables fixed, one gets

$$\sum_I t_I \cdot p(\cdots) + q(x_1, \ldots, x_n) = \sum_I t_I \cdot p(\cdots) = p(\cdots),$$

that is, the evaluation of p for the chosen fixed variables. Following the terminology of [2], p is called the *superpoly* of I in f. A *cube* t_I is called a *maxterm* if and only if its superpoly p has degree 1 (i.e., is linear but not a constant). The polynomial f is called the *master polynomial.*

Given access to a cryptographic function with public and secret variables, the attacker has to recover the secret key variables. Key recovery is achieved in two steps, a preprocessing and an online phase, which are described below.

Preprocessing. One first finds sufficiently many maxterms t_I of the master polynomial. For each maxterm, one computes the coefficients of the secret variables in the symbolic representation of the linear superpoly p. That is, one *reconstructs* the ANF of the superpoly of each t_I. Reconstruction is achieved via probabilistic linearity tests [13], to check that a superpoly is linear, and to identify which variables it contains. The maxterms and superpolys are not key-dependent, thus they need to be computed only once per master polynomial.

The main challenge of the cube attack is to find maxterms. We propose the following simple preprocessing heuristic: one randomly chooses a subset I of k public variables. Thereafter one uses a linearity test to check whether p is linear. If the subset I is too small, the corresponding superpoly p is likely to be a nonlinear function in the secret variables, and in this case the attacker adds a public variable to I and repeats the process. If I is too large, the sum will be a constant function, and in this case he drops one of the public variables from I and repeats the process. The correct choice of I is the borderline between these cases, and if it does not exist the attacker retries with a different initial I.

Online Phase. Once sufficiently many maxterms and the ANF of their superpolys are found, preprocessing is finished and one performs the online phase. Now the secret variables are fixed: one evaluates the superpoly's p by summing $f(x)$ over all the values of the corresponding maxterm, and gets as a result a linear combination of the key bits (because the superpolys are linear). The public variables that are not in the maxterm should be set to a fixed value, and to the same value as set in the preprocessing phase.

Assuming that the degree of the master polynomial is d, each sum requires at most 2^{d-1} evaluations of the derived polynomials (which the attacker obtains via a chosen plaintext attack). Once enough linear superpolys are found, the key can be recovered by simple linear algebra techniques.

1.2 MD6

Rivest presented the hash function MD6 [14,15] as a candidate for NIST's hash competition[4]. MD6 shows originality in both its operation mode—a parametrized quadtree [16]—and its compression function, which repeats hundreds of times a simple combination of XOR's, AND's and shift operations: the r-round compression function of MD6 takes as input an array $A_0, \ldots, A_{88}$ of 64-bit words, recursively computes $A_{89}, \ldots, A_{16r+88}$, and outputs the 16 words $A_{16r+73}, \ldots, A_{16r+88}$:

```
**for** $i = 89, \ldots, 16r + 88$
    $x \leftarrow S_i \oplus A_{i-17} \oplus A_{i-89} \oplus (A_{i-18} \wedge A_{i-21}) \oplus (A_{i-31} \wedge A_{i-67})$
    $x \leftarrow x \oplus (x \gg r_i)$
    $A_i \leftarrow x \oplus (x \ll \ell_i)$
**return** $A_{16r+73,\ldots,16r+88}$
```

A *step* is one iteration of the above loop, a *round* is a sequence of 16 steps. The values S_i, r_i, and ℓ_i are step-dependent constants (see Appendix A). MD6 generates the input words $A_0, \ldots, A_{88}$ as follows:

1. $A_0, \ldots, A_{14}$ contain constants (fractional part of $\sqrt{6}$; 960 bits)
2. $A_{15}, \ldots, A_{22}$ contain a key (512 bits)
3. A_{23}, A_{24} contain parameters (key length, root bit, digest size, etc.; 128 bits)
4. $A_{25}, \ldots, A_{88}$ contain the data to be compressed (message block or chain value; 4096 bits)

The proposed instances of MD6 perform at least 80 rounds (1280 steps) and at most 168 (2688 steps). Resistance to "standard" differential attacks for collision finding is proven for up to 12 rounds. The designers of MD6 could break at most 12 rounds with high complexity using SAT-solvers.

The compression function of MD6 can be seen as a device composed of 64 nonlinear feedback shift registers (NFSR's) and a linear combiner: during a step the 64 NFSR's are clocked in parallel, then linearly combined. The AND operators ($\wedge$) progressively increase nonlinearity, and the shift operators provide wordwise diffusion. This representation will make our attacks easier to understand.

1.3 Trivium

The stream cipher Trivium was designed by De Cannière and Preneel [3] and submitted as a candidate to the eSTREAM project in 2005. Trivium was eventually chosen as one of the four hardware ciphers in the eSTREAM portofolio[5]. Reduced variants of Trivium underwent several attacks [7–9, 17–21], including cube attacks [2].

Trivium takes as input a 80-bit key and a 80-bit IV, and produces a keystream after 1152 rounds of initialization. Each round corresponds to clocking three feedback shift registers, each one having a quadratic feedback polynomial. The best result on Trivium is a cube attack [2] on a reduced version with 767 initialization rounds instead of 1152.

[4]See `http://www.nist.gov/hash-competition`

[5]See `http://www.ecrypt.eu.org/stream/`

1.4 The Contributions of This Paper

First we apply cube attacks to keyed versions of the compression function of MD6. The MD6 team managed to break up to 12 rounds using a high complexity attack based on SAT solvers. In this paper we show how to break the same 12 round version and recover the full 128-bit key with trivial complexity using a cube attack, even under the assumption that the attacker does not know anything about its design (i.e., assuming that the algorithm had not been published and treating the function as a black box polynomial). By exploiting the known internal structure of the function, we can improve the attack and recover the 128-bit key of a 14-round MD6 function in about 2^{22} operations, which take less than a minute on a single PC. This is the best key recovery attack announced so far on MD6.

Then we introduce the new notion of *cube tester*, which combines the cube attack with efficient property-testers, and can be used to mount distinguishers or to detect nonrandomness in cryptographic primitives. Cube testers are flexible attacks that are adaptable to the primitive attacked. Some cube testers don't require the function attacked to have a low degree, but just to satisfy some testable property with significantly higher (or lower) probability than a random function. To the best of our knowledge, this is one of the first explicit applications of property-testing to cryptanalysis.

Applying cube testers to MD6, we can detect nonrandomness in reduced versions with up to 18 rounds in just 2^{17} time. In a variant of MD6 in which all the step constants S_i are zero, we could detect nonrandomness up to 66 rounds using 2^{24} time. Applied to Trivium, cube testers give distinguishers on up to 790 in time 2^{30}, and detect nonrandomness on up to 885 rounds in 2^{27}. Table 1 summarizes our results on MD6 and Trivium, comparing them with the previous attacks .

As Table 1 shows, all our announced complexities are quite low, and presumably much better results can be obtained if we allow a complexity bound of 2^{50} (which is currently practical on a large network of PC's) or even 2^{80} (which may become practical in the future). However, it is very difficult to estimate the performance of cube attacks on larger versions without actually finding the best choice of cube variables, and thus our limited experimental resources allowed us to discover only low complexity attacks. On the other hand, all our announced attacks are fully tested and verified, whereas other types of algebraic attacks are often based on the conjectured independence of huge systems of linear equations, which is impossible to verify in a realistic amount of time.

2 Key Recovery on MD6

2.1 Method

We describe the attack on reduced-round variants of a basic keyed version of the MD6 compression function. The compression function of the basic MD6 keyed version we tested uses a key of 128 bits, and outputs 5 words. Initially, we used

Table 1. Summary of the best known attacks on MD6 and Trivium ("$\surd$" designates the present paper).

#Rounds	Time	Attack	Authors
		MD6	
12	hours	inversion	[15]
14	2^{22}	key recovery	$\surd$
18	2^{17}	nonrandomness	$\surd$
$66^{\star}$	2^{24}	nonrandomness	$\surd$
		Trivium	
736	2^{33}	distinguisher	[7]
$736^{\diamond}$	2^{30}	key-recovery	[2]
$767^{\diamond}$	2^{36}	key-recovery	[2]
772	2^{24}	distinguisher	$\surd$
785	2^{27}	distinguisher	$\surd$
790	2^{30}	distinguisher	$\surd$
842	2^{24}	nonrandomness	$\surd$
885	2^{27}	nonrandomness	$\surd$

*: for a modified version where $S_i = 0$
$^{\diamond}$: cost excluding precomputation

the basic cube attack techniques that treat the compression function as a black box, and were able to efficiently recover the key for up to 12 rounds. We then used the knowledge of the internal structure of the MD6 compression function to improve on these results. The main idea of the improved attack is to choose the public variables in the cube that we sum over so that they do not mix with the key in the initial mixing rounds. In addition, the public variables that do not belong to the cube are assigned predefined constant values that limit the diffusion of the private variables and the cube public variables in the MD6 array for as many rounds as possible. This reduces the degree of the polynomials describing the output bits as functions in the private variables and the cube public variables, improving the performance of the cube attack.

The improved attack is based on the observation that in the feedback function, A_i depends on A_{i-17}, A_{i-89}, A_{i-18}, A_{i-31} and A_{i-67}. However, since A_{i-18} is ANDed with A_{i-21}, the dependency of A_i on A_{i-18} can be eliminated regardless of its value, by zeroing A_{i-21} (assuming the value of A_{i-21} can be controlled by the attacker). Similarly, dependencies on A_{i-21}, A_{i-31} or A_{i-67} can be eliminated by setting the corresponding ANDed word to zero. On the other hand, removing the linear dependencies of A_i on A_{i-17} or A_{i-89} is not possible if their value is unknown (e.g. for private variables), and even if their values are known (e.g. for public variables), the elimination introduces another dependency, which may contribute to the diffusion of the cube public variables (for example it is possible to remove the dependency of A_i on A_{i-89} by setting A_{i-17} to the same value, introducing the dependency of A_{i-17} on A_{i-89}).

These observations lead to the conclusion that the attacker can limit the diffusion of the private variables by removing as many quadratic dependencies of the array variables on the private variables as possible. The basic MD6 keyed version that we tested uses a 2-word (128-bit) key, which is initially placed in A_{15} and A_{16}. Note that the MD6 mode of operation dedicates a specific part of the input to the key in words $A_{15}, \ldots, A_{22}$ (512 bits in total).

Table 2 describes the diffusion of the key into the MD6 compression function array up to step 189 (the index of the first outputted word is 89).

In contrast to the predefined private variable indexes, the attacker can choose the indexes of the cube public variables, and improve the complexity of the attack by choosing such cube public variables that diffuse linearly to the MD6 array only at the later stages of the mixing process. Quadratic dependencies of an array word on cube public variables can be eliminated if the attacker can control the value of the array word that is ANDed with the array word containing the cube public variables. It is easy to verify that the public variable word that is XORed back to the MD6 array at the latest stage of the mixing process is A_{71}, which is XORed in step 160 to A_{160}. Thus, the array word with index 71 and words with index just under 71, seem to be a good choice for the cube public variables. Exceptions are A_{68} and A_{69} which are mixed with the key in steps 135 and 136 and should be zeroed. We tested several cubes, and the best preprocessing results were obtained by choosing cube indexes from A_{65}. One of the reason that A_{65} gives better results than several other words (e.g. A_{71}) is that it is ANDed with just 2 words before it is XORed again into the array in step 154, whereas A_{71} is ANDed with 4 words before step 170. This gives the attacker more freedom to choose the values of the fixed public variables, and limit the diffusion of the private and cube public variables for more rounds. Table 3 describes the diffusion of A_{65} into the MD6 compression function array up to step 185 (the index of the first outputted word is 89).

2.2 Results

We were able to prevent non-linear mixing of the cube public variables and the private variables for more than 6 MD6 compression function rounds. This was made possible by zeroing all the MD6 array words whose indexes are listed in the third column of Table 2 and Table 3 (ignoring the special "L" values). As described in the previous section, we set the values of several of the 63 attacker controlled words, excluding A_{65} (from which the cube public variables were chosen), to predefined constants that zero the words specified in the third column. Public variables whose value does not affect the values of the listed MD6 array words were set to zero. We were not able to limit the diffusion of the cube public variables and the private variables as much when all the cube public variable indexes were chosen from words other than A_{65}.

We describe the cube attack results on the keyed MD6 version. The results were obtained by running the preprocessing phase of the cube attack with the special parameters describes above. We found many dense maxterms for 13-round MD6, with associated cubes of size 5. Each of the maxterms passed at

Table 2. Diffusion of the private variables into the MD6 compression function array in the initial mixing steps. The third column specifies the MD6 array index of the word that is ANDed with the key-dependent array word index in the step number specified by the first column. The output of step i is inserted into A_i. If the key-dependent array word is diffused linearly, then L is written instead. Note that once a dependency of an MD6 array word on the private variables can be eliminated, it does not appear any more as key-dependent (i.e. we assume that this dependency is eliminated by the attacker).

Step	Key-dependent array index	ANDed index
104	15	L
105	16	L
121	104	L
122	105	L
122	104	101
123	105	102
125	104	107
126	105	108
135	104	68
136	105	69
138	121	L
139	122	L
139	121	118
140	122	119
142	121	124
143	122	125
152	121	85
153	122	86
155	138	L
156	139	L
156	138	135
157	139	136
159	138	141
160	139	142
169	138	102
170	139	103
171	104	140
172	105	141
172	155	L
173	156	L
173	155	152
174	156	153
176	155	158
177	156	159
186	155	119
187	156	120
187	121	157
188	122	158

Table 3. Diffusion of A_{65} into the MD6 compression function array in the initial mixing rounds (if the key-dependent array word is diffused linearly, then L is written instead)

Step	A_{65}-dependent array index	Multiplicand index
96	65	29
132	65	101
154	65	L
171	154	L
172	154	151
175	154	157
185	154	118

least 100 linearity tests, thus the maxterm equations are likely to be correct for most keys. During the online phase, the attacker evaluates the superpolys by summing over the cubes of size 5. This requires a total of about 2^{12} chosen IVs. The total complexity of the attack is thus no more than 2^{12}.

We were able to find many constant superpolys for 14 rounds of MD6, with associated cubes of size 7. However, summing on cubes of size 6 gives superpolys of high degree in the key bits. In order to further eliminate most (but not all) high degree terms from the superpolys obtained by summing on cubes of size 6, we added more public variable indexes from words other than A_{65}. The best results were obtained by choosing the remaining indexes from A_{32}, A_{33}, A_{49} and A_{50} (which are directly XORed with key bits in steps 121, 122, 138 and 139). Using this approach, we found many dense maxterms for 14-round MD6, with associated cubes of size 15. Some of these results are listed in Table 5 (Appendix A), many more linearly independent maxterms can be easily obtained by choosing other cube indexes from the same words listed in Table 5. During the online phase, the attacker evaluates the superpolys by summing over the cubes of size 15. This requires a total of about 2^{22} chosen IVs. The total complexity of the attack is thus no more than 2^{22}. In fact every IV gives many maxterms, so the required total of chosen IVs is lower than 2^{22}, and the total complexity of the attack is less than 2^{22}.

We were able to find many constant superpolys for 15 rounds of MD6, with associated cubes of size 14. We were not able to find low degree superpolys for 15-round MD6. However, it seems likely that low degree equation for 15-round MD6 can be obtained using approaches similar to the one we used to recover the key for 14-round MD6. Hence we believe that cube attacks can efficiently recover the key for 15-round MD6. Furthermore, we believe that cube key recovery attacks will remain faster than exhaustive search for 18-19 MD6 rounds.

3 Cube Testers

3.1 Definitions

Recall that $\mathcal{F}_n$ denotes the set of all functions mapping $\{0,1\}^n$ to $\{0,1\}$, $n > 0$. For a given n, a *random function* is a random element of $\mathcal{F}_n$ (we have $|\mathcal{F}_n| = 2^{2^n}$). In the ANF of a random function, each monomial (and in particular, the highest degree monomial $x_1 \cdots x_n$) appears with probability 1/2, hence a random function has maximal degree of n with probability 1/2. Similarly, it has degree $(n-2)$ or less with probability $1/2^{n+1}$. Note that the explicit description of a random function can be directly expressed as a circuit with, in average, 2^{n-1} gates (AND and XOR), or as a string of 2^n bits where each bit is the coefficient of a monomial (encoding the truth table also requires 2^n bits, but hides the algebraic structure).

Informally, a *distinguisher* for a family $\mathcal{F} \subsetneq \mathcal{F}_n$ is a procedure that, given a function f randomly sampled from $\mathcal{F}^\star \in \{\mathcal{F}, \mathcal{F}_n\}$, efficiently determines which one of these two families was chosen as $\mathcal{F}^\star$. A family $\mathcal{F}$ is *pseudorandom* if and only if there exists no efficient distinguisher for it. In practice, e.g. for hash functions or ciphers, a family of functions is defined by a k-bit parameter of the function, randomly chosen and unknown to the adversary, and the function is considered broken (or, at least, "nonrandom") if there exists a distinguisher making significantly less than 2^k queries to the function. Note that a distinguisher that runs in exponential time in the key may be considered as "efficient" in practice, e.g. 2^{k-10}.

We would like to stress the terminology difference between a *distinguisher* and the more general detection of *pseudorandomness*, when speaking about cryptographic algorithms; the former denotes a distinguisher (as defined above) where the parameter of the family of functions is the cipher's *key*, and thus can't be modified by the adversary through its queries; the latter considers part of the key as a public input, and assumes as secret an arbitrary subset of the input (including the input bits that are normally public, like IV bits). The detection of nonrandomness thus does not necessarily correspond to a realistic scenario. Note that related-key attacks are captured by neither one of those scenarios.

To distinguish $\mathcal{F} \subsetneq \mathcal{F}_n$ from $\mathcal{F}_n$, cube testers partition the set of public variables $\{x_1, \ldots, x_n\}$ into two complementary subsets:

- *cube variables* (CV)
- *superpoly variables* (SV)

We illustrate these notions with the example from §1.1: recall that, given

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3 \ ,$$

we considered the *cube* x_1x_2 and called $(x_3 + x_4)$ its *superpoly*, because

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2(x_3 + x_4) + x_3 \ .$$

Here the cube variables (CV) are x_1 and x_2, and the superpoly variables (SV) are x_3 and x_4. Therefore, by setting a value to x_3 and x_4, e.g. $x_3 = 0$, $x_4 = 1$, one can compute $(x_3 + x_4) = 1$ by summing $f(x_1, x_2, x_3, x_4)$ for all possibles choices of (x_1, x_2). Note that it is not required for a SV to actually appear in the superpoly of the maxterm. For example, if $f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3$, then the superpoly of x_1x_2 is x_3, but the SV's are both x_3 and x_4.

Remark. When f is, for example, a hash function, not all inputs should be considered as variables, and not all Boolean components should be considered as outputs, for the sake of efficiency. For example if f maps 1024 bits to 256 bits, one may choose 20 CV and 10 SV and set a fixed value to the other inputs. These fixed inputs determine the coefficient of each monomial in the ANF with CV and SV as variables. This is similar to the preprocessing phase of key-recovery cube attacks, where one has access to all the input variables. Finally, for the sake of efficiency, one may only evaluate the superpolys for 32 of the 256 Boolean components of the output.

3.2 Examples

Cube testers distinguish a family of functions from random functions by testing a property of the superpoly for a specific choice of CV and SV. This section introduces this idea with simple examples. Consider

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3$$

and suppose we choose CV x_3 and x_4 and SV x_1 and x_2, and evaluate the superpoly of x_3x_4:

$$f(x_1, x_2, 0, 0) + f(x_1, x_2, 0, 1) + f(x_1, x_2, 1, 0) + f(x_1, x_2, 1, 1) = 0 \ ,$$

This yields zero for any $(x_1, x_2) \in \{0, 1\}^2$, i.e. the superpoly of x_3x_4 is zero, i.e. none of the monomials x_3x_4, $x_1x_3x_4$, $x_2x_3x_4$, or $x_1x_2x_3x_4$ appears in f. In comparison, in a random function the superpoly of x_3x_4 is null with probability only 1/16, which suggests that f was not chosen at random (indeed, we chose it particularly sparse, for clarity). Generalizing the idea, one can deterministically test whether the superpoly of a given maxterm is constant, and return "random function" if and only if the superpoly is not constant. This is similar to the test used in [7].

Let $f \in \mathcal{F}_n$, $n > 4$. We present a probabilistic test that detects the presence of monomials of the form $x_1x_2x_3x_i \dots x_j$ (e.g. $x_1x_2x_3$, $x_1x_2x_3x_n$, etc.):

1. choose a random value of $(x_4, \dots, x_n) \in \{0, 1\}^{n-4}$
2. sum $f(x_1, \dots, x_n)$ over all values of (x_1, x_2, x_3), to get

$$\sum_{(x_1,x_2,x_3)\in\{0,1\}^3} f(x_1, \dots, x_n) = p(x_4, \dots, x_n)$$

where p is a polynomial such that

$$f(x_1, \dots, x_n) = x_1x_2x_3 \cdot p(x_4, \dots, x_n) + q(x_1, \dots, x_n)$$

where the polynomial q contains no monomial with $x_1x_2x_3$ as a factor in its ANF

3. repeat the two previous steps N times, recording the values of $p(x_4, \ldots, x_n)$

If f were a random function, it would contain at least one monomial of the form $x_1x_2x_3x_i \ldots x_j$ with high probability; hence, for a large enough number of repetitions N, one would record at least one nonzero $p(x_4, \ldots, x_n)$ with high probability. However, if no monomial of the form $x_1x_2x_3x_i \ldots x_j$ appears in the ANF, $p(x_4, \ldots, x_n)$ always evaluates to zero.

3.3 Building on Property Testers

Cube testers combine an efficient property tester on the superpoly, which is viewed either as a polynomial or as a mapping, with a statistical decision rule. This section gives a general informal definition of cube testers, starting with basic definitions. A *family tester* for a family of functions $\mathcal{F}$ takes as input a function f of same domain $\mathcal{D}$ and tests if f is close to $\mathcal{F}$, with respect to a bound ϵ on the distance

$$\delta(f, \mathcal{F}) = \min_{g \in \mathcal{F}} \frac{|\{x \in \mathcal{D}, f(x) \neq g(x)\}|}{|\mathcal{D}|} \ .$$

The tester accepts if $\delta(f, \mathcal{F}) = 0$, rejects with high probability if f and $\mathcal{F}$ are not ϵ-close, and behaves arbitrarily otherwise. Such a test captures the notion of property-testing, when a property is defined by belonging to a family of functions $\mathcal{P}$; a *property tester* is thus a family tester for a property $\mathcal{P}$.

Suppose one wishes to distinguish a family $\mathcal{F} \subsetneq \mathcal{F}_n$ from $\mathcal{F}_n$, i.e., given a random $f \in \mathcal{F}^\star$, to determine whether $\mathcal{F}^\star$ is $\mathcal{F}$ or $\mathcal{F}_n$ (for example, in Trivium, $\mathcal{F}$ may be a superpoly with respect to CV and SV in the IV bits, such that each $f \in \mathcal{F}$ is computed with a distinct key). Then if $\mathcal{F}$ is efficiently testable (see [22,23]), then one can use directly a family tester for $\mathcal{F}$ on f to distinguish it from a random function.

Cube testers detect nonrandomness by applying property testers to superpolys: informally, as soon as a superpoly has some "unexpected" property (that is, is anormally structured) it is identified as nonrandom. Given a testable property $\mathcal{P} \subsetneq \mathcal{F}_n$, cube testers run a tester for $\mathcal{P}$ on the superpoly function f, and use a statistical decision rule to return either "random" or "nonrandom". The decision rule depends on the probabilities $|\mathcal{P}|/|\mathcal{F}_n|$ and $|\mathcal{P} \cap \mathcal{F}|/|\mathcal{F}|$ and on a margin of error chosen by the attacker. Roughly speaking, a family $\mathcal{F}$ will be distinguishable from $\mathcal{F}_n$ using the property $\mathcal{P}$ if

$$\left| \frac{|\mathcal{P}|}{|\mathcal{F}_n|} - \frac{|\mathcal{P} \cap \mathcal{F}|}{|\mathcal{F}|} \right|$$

is non-negligible. That is, the tester will determine whether f is significantly closer to $\mathcal{P}$ than a random function. Note that the dichotomy between structure (e.g. testable properties) and randomness has been studied in [24].

3.4 Examples of Testable Properties

Below, we give examples of efficiently testable properties of the superpoly, which can be used to build cube testers (see [23] for a general characterization of efficiently testable properties). We let C be the size of CV, and S be the size of SV; the complexity is given as the number of evaluations of the tested function f. Note that each query of the tester to the superpoly requires 2^C queries to the target cryptographic function. The complexity of any property tester is thus, even in the best case, exponential in the number of CV.

Balance. A random function is expected to contain as many zeroes as ones in its truth table. Superpolys that have a strongly unbalanced truth table can thus be distinguished from random polynomials, by testing whether it evaluates as often to one as to zero, either deterministically (by evaluating the superpoly for each possible input), or probabilistically (over some random subset of the SV). For example, if CV are $x_1, \dots, x_C$ and SV are $x_{C+1}, \dots, x_n$, the deterministic balance test is

```
1. $c \leftarrow 0$
2. **for** all values of $(x_{C+1}, \dots, x_n)$
3.     compute
           $$p(x_{C+1}, \dots, x_n) = \sum_{(x_1, \dots, x_C)} f(x_1, \dots, x_n) \; \in \{0,1\}$$
4.     $c \leftarrow c + p(x_{C+1}, \dots, x_n)$
5. **return** $\mathcal{D}(c) \in \{0,1\}$
```

where $\mathcal{D}$ is some decision rule. A probabilistic version of the test makes $N < 2^S$ iterations, for random distinct values of $(x_{C+1}, \dots, x_n)$. Complexity is respectively 2^n and $N \cdot 2^C$.

Constantness. A particular case of balance test considers the "constantness" property, i.e. whether the superpoly defines a constant function; that is, it detects either that f has maximal degree strictly less than C (null superpoly), or that f has maximal degree exactly C (superpoly equals the constant 1), or that f has degree strictly greater than C (non-constant superpoly). This is equivalent to the maximal degree monomial test used in [7], used to detect nonrandomness in 736-round Trivium.

Low Degree. A random superpoly has degree at least $(S-1)$ with high probability. Cryptographic functions that rely on a low-degree function, however, are likely to have superpolys of low degree. Because it closely relates to probabilistically checkable proofs and to error-correcting codes, low-degree testing has been well studied; the most relevant results to our concerns are the tests for Boolean functions in [25, 26]. The test by Alon et al. [25], for a given degree d, queries the function at about $d \cdot 4^d$ points and always accepts if the ANF of the function has degree at most k, otherwise it rejects with some bounded error probability. Note that, contrary to the method of ANF reconstruction (exponential in S), the complexity of this algorithm is *independent of the number of variables*. Hence, cube testers based on this low-degree test have complexity which is independent of the number of SV's.

Presence of Linear Variables. This is a particular case of the low-degree test, for degree $d = 1$ and a single variable. Indeed, the ANF of a random function contains a given variable in at least one monomial of degree at least two with probability close to 1. One can thus test whether a given superpoly variable appears only linearly in the superpoly, e.g. for x_1 using the following test similar to that introduced in [13]:

```
1. pick random $(x_2, \ldots, x_S)$
2. **if** $p(0, x_2, \ldots, x_S) = p(1, x_2, \ldots, x_S)$
3.     **return** nonlinear
4. repeat steps 1 to 3 $N$ times
5. **return** linear
```

This test answers correctly with probability about $1 - 2^{-N}$, and computes $N \cdot 2^{C+1}$ times the function f. If, say, a stream cipher is shown to have an IV bit linear with respect to a set of CV in the IV, independently of the choice of the key, then it directly gives a distinguisher.

Presence of Neutral Variables. Dually to the above linearity test, one can test whether a SV is neutral in the superpoly, that is, whether it appears in at least one monomial. For example, the following algorithm tests the neutrality of x_1, for $N \leq 2^{S-1}$:

```
1. pick random $(x_2, \ldots, x_S)$
2. **if** $p(0, x_2, \ldots, x_S) \neq p(1, x_2, \ldots, x_S)$
3.     **return** not neutral
4. repeat steps 1 to 3 $N$ times
5. **return** neutral
```

This test answers correctly with probability about $1 - 2^{-N}$ and runs in time $N \cdot 2^C$. For example, if x_1, x_2, x_3 are the CV and x_4, x_5, x_6 the SV, then x_6 is neutral with respect to $x_1x_2x_3$ if the superpoly $p(x_4, x_5, x_6)$ satisfies $p(x_4, x_5, 0) = p(x_4, x_5, 1)$ for all values of (x_4, x_5). A similar test was implicitly used in [9], via the computation of a *neutrality measure*.

Remarks. Except low degree and constantness, the above properties do not require the superpoly to have a low degree to be tested. For example if the maxterm x_1x_2 has the degree-5 superpoly

$$x_3x_5x_6 + x_3x_5x_6x_7x_8 + x_5x_8 + x_9 \ ,$$

then one can distinguish this superpoly from a random one either by detecting the linearity of x_9 or the neutrality of x_4, with a cost independent on the degree. In comparison, the cube tester suggested in [2] required the degree to be bounded by d such that 2^d is feasible.

Note that the cost of detecting the property during the preprocessing is larger than the cost of the on-line phase of the attack, given the knowledge of the property. For example, testing that x_1 is a neutral variable requires about $N \cdot 2^C$ queries to the function, but once this property is known, 2^C queries are sufficient to distinguish the function from a random one with high probability.

Finally, note that tests based on the nonrandom distribution of the monomials [4–6] are not captured by our definition of cube testers, which focus on high-degree terms. Although, in principle, there exist cases where the former tests would succeed while cube testers would fail, in practice a weak distribution of lower-degree monomials rarely comes with a good distribution of high-degree ones, as results in [7] and of ourselves suggest.

4 Cube Testers on MD6

We use cube testers to detect nonrandom properties in reduced-round versions of the MD6 compression function, which maps the 64-bit words $A_0, \ldots, A_{88}$ to $A_{16r+73}, \ldots, A_{16r+88}$, with r the number of rounds. From the compression function $f : \{0,1\}^{64\times 89} \mapsto \{0,1\}^{64\times 16}$, our testers consider families of functions $\{f_m\}$ where a random $f_i : \{0,1\}^{64\times 89-k} \mapsto \{0,1\}^{64\times 16}$ has k input bits set to a random k-bit string. The attacker can thus query f_i, for a randomly chosen key i, on $(64 \times 89 - k)$-bit inputs.

The key observations leading to our improved attacks on MD6 are that:

1. input words appear either linearly (as A_{i-89} or A_{i-17}) or nonlinearly (as A_{18}, A_{21}, A_{31}, or A_{67}) within a step
2. words $A_0, \ldots, A_{21}$ are input once, $A_{22}, \ldots, A_{57}$ are input twice, $A_{58}, \ldots, A_{67}$ are input three times, A_{68}, A_{69}, A_{70} four times, A_{71} five times, and $A_{72}, \ldots, A_{88}$ six times
3. all input words appear linearly at least once ($A_0, \ldots, A_{71}$), and at most twice ($A_{72}, \ldots, A_{88}$)
4. A_{57} is the last word input (at step 124, i.e. after 2 rounds plus 3 steps)
5. A_{71} is the last word input linearly (at step 160, i.e. after 4 rounds plus 7 steps)
6. differences in a word input nonlinearly are "absorbed" if the second operand is zero (e.g. $A_{i-18} \wedge A_{i-21} = 0$ if A_{i-18} is zero, for any value of A_{i-21})

Based on the above observations, the first attack (A) makes only black-box queries to the function. The second attack (B) can be seen as a kind of related-key attack, and is more complex and more powerful. Our best attacks, in terms of efficiency and number of rounds broken, were obtained by testing the *balance* of superpolys.

4.1 Attack A

This attack considers CV, SV, and secret bits in A_{71}: the MSB's of A_{71} contain the CV, the LSB's contain the 30 secret bits, and the 4 bits "in the middle" are the SV. The other bits in A_{71} are set to zero. To minimize the density and the degree of the ANF, we set $A_i = S_i$ for $i = 0, \dots, 57$ in order to eliminate the constants S_i from the expressions, and set $A_i = 0$ for $i = 58, \dots, 88$ in order to eliminate the quadratic terms by "absorbing" the nonzero $A_{22}, \dots, A_{57}$ through AND's with zero values.

The attack exploits the fact that A_{71} is the last word input linearly. We set initial conditions on the message such that modifications in A_{71} are only effective at step 160, and so CV and SV are only introduced (linearly) at step 160: in order to absorb A_{71} before step 160, one needs $A_{68} = A_{74} = A_{35} = A_{107} = 0$, respectively for steps 89, 92, 102, and 138.

Given the setup above, the attack evaluates the balance of the superpoly for each of the 1024 output components, in order to identify superpolys that are constant for a large majority of inputs (SV). These superpolys may be either constants, or unbalanced nonlinear functions. Results for reduced and modified MD6 are given in subsequent sections.

4.2 Attack B

This attack considers CV, SV, and secret bits in A_{54}, at the same positions as in Attack A. Other input words are set by default to S_i for $A_0, \dots, A_{47}$, and to zero otherwise.

The attack exploits the fact that A_{54} and A_{71} are input linearly only once, and that both directly interact with A_{143}. We set initial conditions on the message such that CV and SV are only effective at step 232. Here are the details of this attack:

- step 143: input variables are transfered linearly to A_{143}
- step 160: A_{143} is input linearly; to cancel it, and thus to avoid the introduction of the CV and SV in the ANF, one needs $A_{71} = S_{160} \oplus A_{143}$
- step 92: A_{71} is input nonlinearly; to cancel it, in order to make A_{138} independent of A_{143}, we need $A_{74} = 0$
- step 138: A_{71} is input nonlinearly; to cancel it, one needs $A_{107} = 0$
- step 161: A_{143} is input nonlinearly; to cancel it, one needs $A_{140} = 0$
- step 164: A_{143} is input nonlinearly; to cancel it, one needs $A_{146} = 0$
- step 174: A_{143} is input nonlinearly; to cancel it, one needs $A_{107} = 0$ (as for step 138)
- step 210: A_{143} is input nonlinearly; to cancel it, one needs $A_{179} = 0$
- step 232: A_{143} is input linearly, and introduces the CV and SV linearly into the ANF

To satisfy the above conditions, one has to choose suitable values of A_1, A_{18}, A_{51}, A_{57}, A_{74}. These values are constants that do not depend on the input in A_{54}.

Given the setup above, the attack evaluates the balance of the superpoly for each of the 1024 output components, in order to identify superpolys that are constant for large majority of inputs (SV). Results for reduced and modified MD6 are given in §4.3.

4.3 Results

In this subsection we report the results we obtained by applying attacks A and B to reduced versions of MD6, and to a modified version of MD6 that sets all the constants S_i to zero. Recall that by using C CV's, the complexity of the attack is about 2^C computations of the function. We report results for attacks using at most 20 CV (i.e. doable in less than a minute on a single PC):

– with *attack A*, we observed strong imbalance after 15 rounds, using 19 CV. More precisely, the Boolean components corresponding to the output bits in A_{317} and A_{325} all have (almost) constant superpoly. When all the S_i constants are set to 0, we observed that all the outputs in A_{1039} and A_{1047} have (almost) constant superpoly, i.e. we can break 60 rounds of this modified MD6 version using only 14 CV's.
– with *attack B*, we observed strong imbalance after 18 rounds, using 17 CV's. The Boolean components corresponding to the output bits in A_{368} and A_{376} all have (almost) constant superpoly. When $S_i = 0$, using 10 CV's, one finds that all outputs in A_{1114} and A_{1122} have (almost) constant superpoly, i.e. one breaks 65 rounds. Pushing the attack further, one can detect nonrandomness after 66 rounds, using 24 CV's.

The difference of results between the original MD6 and the modified case in which $S_i = 0$ comes from the fact that a zero S_i makes it possible to keep a sparse state during many rounds, whereas a nonzero S_i forces the introduction of nonzero bits in the early steps, thereby quickly increasing the density of the implicit polynomials, which indirectly facilitates the creation of high degree monomials.

5 Cube Testers on Trivium

Observations in [2, Tables 1,2,3] suggest nonrandomness properties detectable in time about 2^{12} after 684 rounds, in time 2^{24} after 747 rounds, and in time 2^{30} after 774 rounds. However, a distinguisher cannot be directly derived because the SV used are in the key, and thus cannot be chosen by the attacker in an attack where the key is fixed.

5.1 Setup

We consider families of functions defined by the secret key of the cipher, and where the IV corresponds to public variables. We first used the 23-variable index sets identified in [27, Table 2]; even though we have not tested all entries, we obtained the best results using the IV bits (starting from zero)

$$\{3, 4, 6, 9, 13, 17, 18, 21, 26, 28, 32, 34, 37, 41, 47, 49, 52, 58, 59, 65, 70, 76, 78\} .$$

For this choice of CV, we choose 5 SV, either

– in the IV, at positions $0, 1, 2, 35, 44$ (to have a distinguisher), or
– in the key, at positions $0, 1, 2, 3, 4$ (to detect nonrandomness)

For experiments with 30 CV, we use another index set discovered in [27]:

$$\{1, 3, 6, 12, 14, 18, 22, 23, 24, 26, 30, 32, 33, 35, 36, 39, 40, 44, 47, 49, 50, 53, 59, 60, 61, 66, 68, 69, 72, 75\}\ .$$

IV bits that are neither CV nor SV are set to zero, in order to minimize the degree and the density of the polynomials generated during the first few initialization steps. Contrary to MD6, we obtain the best results on Trivium by testing the presence of *neutral variables*. We look for neutral variables either for a random key, or for the special case of the zero key, which is significantly weaker with respect to cube testers.

In addition to the cubes identified in [27, Table 2], we were able to further improve the results by applying cube testers on carefully chosen cubes, where the indexes are uniformly spread (the distance between neighbors is at least 2). These cubes exploit the internal structure of Trivium, where non linear operations are only performed on consecutive cells. The best results were obtained using the cubes below:

$$\{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 33, 36, 39, 42, 45, 48, 51, 60, 63, 66, 69, 72, 75, 79\}$$
$$\{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 79\}$$
$$\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 79\}$$

5.2 Results

We obtained the following results, by testing the *neutrality* of the SV in the superpoly:

– with 23 CV, and SV in the IV, we found a distinguisher on up to 749 rounds (runtime 2^{23}); SV 0, 1, 2, and 3 are neutral after 749 initialization rounds. Using the zero key, neutral variables are observed after 755 rounds (SV 0, 1 are neutral).
– with 23 CV, and SV in the key, we observed nonrandomness after 758 initialization rounds (SV 1, 2, 3 are neutral). Using the zero key, nonrandomness was observed after 761 rounds (SV 0 is neutral).
– with 30 CV, and SV in the key, we observed nonrandomness after 772 initialization rounds (SV 0, 2, 4 are neutral). Using the zero key, nonrandomness was observed after 782 rounds (SV 2, 3, 4 are neutral).

With the the new chosen cubes we obtain the following results:

– with 24 CV, we observe that the resultant superpoly after 772 initialization rounds is constant, hence we found a distinguisher on up to 772 rounds. Using the neutrality test, for the zero key, we detected nonrandomness over up to 842 rounds (the 4 first key bits are neutral).

– with 27 CV, we observe that the resultant superpoly after 785 initialization rounds is constant, hence we found a distinguisher on up to 785 rounds. Using the neutrality test, for the zero key, we detected nonrandomness over up to 885 rounds (bits 0, 3, and 4 of the key are neutral).
– with 30 CV, we observe that the resultant superpoly after 790 initialization rounds is constant, hence we found a distinguisher for Trivium with up to 790 rounds.

Better results are obtained when the SV's are in the key, not the IV; this is because the initialization algorithm of Trivium puts the key and the IV into two different registers, which make dependency between bits in a same register stronger than between bits in different registers.

In comparison, [7], testing the *constantness* of the superpoly, reached 736 rounds with 33 CV. The observations in [27], obtained by testing the *linearity* of SV in the key, lead to detectable nonrandomness on 748 rounds with 23 CV, and on 771 rounds with 30 CV.

6 Conclusions

We applied cube attacks to the reduced-round MD6 compression function, and could recover a full 128-bit key on 14-round MD6 with a very practical complexity of 2^{22} evaluations. This outperforms all the attacks obtained by the designers of MD6.

Then we introduced the notion of cube tester, based on cube attacks and on property-testers for Boolean functions. Cube testers can be used to mount distinguishers or to simply detect nonrandomness in cryptographic algorithms. Cube testers do not require large precomputations, and can even work for high degree polynomials (provided they have some "unexpected" testable property).

Using cube testers, we detected nonrandomness properties after 18 rounds of the MD6 compression function (the proposed instances have at least 80 rounds). Based on observations in [2], we extended the attacks on Trivium a few more rounds, giving experimentally verified attacks on reduced variants with up to 790 rounds, and detection of nonrandomness on 885 rounds (against 1152 in the full version, and 771 for the best previous attack).

Our results leave several issues open:

1. So far cube attacks have resulted from empirical observations, so that one could only assess the existence of feasible attacks. However, if one could upper-bound the degree of some Boolean component (e.g. of MD6 or Trivium) after a higher number of rounds, then one could predict the existence of observable nonrandomness (and one may build distinguishers based on low-degree tests [25]). The problem is closely related to that of bounding the degree of a nonlinear recursive Boolean sequence which, to the best of our knowledge, has remained unsolved.

2. Low-degree tests may be used for purposes other than detecting nonrandomness. For example, key-recovery cube attacks may be optimized by exploiting low-degree tests, to discover low-degree superpolys, and then reconstruct them. Also, low-degree tests for general fields [28] may be applicable to hash functions based on multivariate systems [29], which remain unbroken over fields larger than GF(2) [30].
3. Our attacks on MD6 detect nonrandomness of reduced versions of the compression function, and even recover a 128-bit key. It would be interesting to extend these attacks to a more realistic scenario, e.g. that would be applicable to the MD6 operation mode, and/or to recover larger keys.
4. One may investigate the existence of cube testers on other primitives that are based on low-degree functions, like RadioGatún, Panama, the stream cipher MICKEY, and on the SHA-3 submissions ESSENCE [31], and Keccak [32]. We propose to use cube attacks and cube testers as a benchmark for evaluating the algebraic strength of primitives based on a low-degree component, and as a reference for choosing the number of rounds. Our preliminary results on Grain-128 outperform all previous attacks, but will be reported later since they are still work in progress.

References

1. Shamir, A.: How to solve it: New techniques in algebraic cryptanalysis. Invited talk at CRYPTO 2008 (2008)
2. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In Joux, A., ed.: EUROCRYPT 2009. LNCS (2009) To appear, see also [27].
3. Cannière, C.D., Preneel, B.: Trivium. In: New Stream Cipher Designs. Volume 4986 of LNCS., Springer (2008) 84–97
4. Filiol, E.: A new statistical testing for symmetric ciphers and hash functions. In Deng, R.H., Qing, S., Bao, F., Zhou, J., eds.: ICICS. Volume 2513 of LNCS., Springer (2002) 342–353
5. Saarinen, M.J.O.: Chosen-IV statistical attacks on eStream ciphers. In Malek, M., Fernández-Medina, E., Hernando, J., eds.: SECRYPT, INSTICC Press (2006) 260–266
6. O'Neil, S.: Algebraic structure defectoscopy. Cryptology ePrint Archive, Report 2007/378 (2007)
7. Englund, H., Johansson, T., Turan, M.S.: A framework for chosen IV statistical analysis of stream ciphers. In Srinathan, K., Rangan, C.P., Yung, M., eds.: INDOCRYPT. Volume 4859 of LNCS., Springer (2007) 268–281
8. Vielhaber, M.: Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. Cryptology ePrint Archive, Report 2007/413 (2007)
9. Fischer, S., Khazaei, S., Meier, W.: Chosen IV statistical analysis for key recovery attacks on stream ciphers. In Vaudenay, S., ed.: AFRICACRYPT. Volume 5023 of LNCS., Springer (2008) 236–245
10. Khazaei, S., Meier, W.: New directions in cryptanalysis of self-synchronizing stream ciphers. In Chowdhury, D.R., Rijmen, V., Das, A., eds.: INDOCRYPT. Volume 5365 of LNCS., Springer (2008) 15–26
11. Lucks, S.: The saturation attack - a bait for Twofish. In Matsui, M., ed.: FSE. Volume 2355 of LNCS., Springer (2001) 1–15

12. Knudsen, L.R.: Truncated and higher order differentials. In Preneel, B., ed.: FSE. Volume 1008 of LNCS., Springer (1994) 196–211
13. Blum, M., Luby, M., Rubinfeld, R.: Self-testing/correcting with applications to numerical problems. In: STOC, ACM (1990) 73–83
14. Rivest, R.L.: The MD6 hash function. Invited talk at CRYPTO 2008 (2008) Slides available at `http://people.csail.mit.edu/rivest/`.
15. Rivest, R.L., Agre, B., Bailey, D.V., Crutchfield, C., Dodis, Y., Fleming, K.E., Khan, A., Krishnamurthy, J., Lin, Y., Reyzin, L., Shen, E., Sukha, J., Sutherland, D., Tromer, E., Yin, Y.L.: The MD6 hash function – a proposal to NIST for SHA-3 (2008) `http://groups.csail.mit.edu/cis/md6/`.
16. Crutchfield, C.Y.: Security proofs for the MD6 hash function mode of operation. Master's thesis, Massachusetts Institute of Technology (2008)
17. Raddum, H.: Cryptanalytic results on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001 (2006)
18. Maximov, A., Biryukov, A.: Two trivial attacks on Trivium. In Adams, C.M., Miri, A., Wiener, M.J., eds.: Selected Areas in Cryptography. Volume 4876 of LNCS., Springer (2007) 36–55
19. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniSat. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/040 (2007)
20. Turan, M.S., Kara, O.: Linear approximations for 2-round Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/008 (2007)
21. Pasalic, E.: Transforming chosen iv attack into a key differential attack: how to break TRIVIUM and similar designs. Cryptology ePrint Archive, Report 2008/443 (2008)
22. Rubinfeld, R., Sudan, M.: Robust characterizations of polynomials with applications to program testing. SIAM J. Comput. **25** (1996) 252–271
23. Kaufman, T., Sudan, M.: Algebraic property testing: the role of invariance. In Ladner, R.E., Dwork, C., eds.: STOC, ACM (2008) 403–412
24. Tao, T.: The dichotomy between structure and randomness, arithmetic progressions, and the primes. In: International Congress of Mathematicians, European Mathematical Society (2006) 581–608
25. Alon, N., Kaufman, T., Krivelevich, M., Litsyn, S., Ron, D.: Testing low-degree polynomials over GF(2). In Arora, S., Jansen, K., Rolim, J.D.P., Sahai, A., eds.: RANDOM-APPROX. Volume 2764 of LNCS., Springer (2003) 188–199
26. Samorodnitsky, A.: Low-degree tests at large distances. In Johnson, D.S., Feige, U., eds.: STOC, ACM (2007) 506–515
27. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. Cryptology ePrint Archive, Report 385 (2008) version 20080914:160327.
28. Kaufman, T., Ron, D.: Testing polynomials over general fields. In: FOCS, IEEE Computer Society (2004) 413–422
29. Billet, O., Robshaw, M.J.B., Peyrin, T.: On building hash functions from multivariate quadratic equations. In Pieprzyk, J., Ghodosi, H., Dawson, E., eds.: ACISP. Volume 4586 of LNCS., Springer (2007) 82–95
30. Aumasson, J.P., Meier, W.: Analysis of multivariate hash functions. In Nam, K.H., Rhee, G., eds.: ICISC. Volume 4817 of LNCS., Springer (2007) 309–323
31. Martin, J.W.: ESSENCE: A candidate hashing algorithm for the NIST competition. Submission to NIST (2008)
32. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak specifications. Submission to NIST (2008) `http://keccak.noekeon.org/`.

A Details on MD6

The word S_i is a round-dependent constant: during the first round (i.e., the first 16 steps) $S_i = \texttt{0123456789abcdef}$, then at each new round it is updated as

$$S_i \leftarrow (S_0 \ll 1) \oplus (S_0 \gg 63) \oplus (S_{i-1} \wedge \texttt{7311c2812425cfa}).$$

The shift distances r_i and ℓ_i are step-dependent constants, see Table 4.

Table 4. Distances of the shift operators used in MD6, as function of the step index within a round.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_i	10	5	13	10	11	12	2	7	14	15	7	13	11	7	6	12
ℓ_i	11	24	9	16	15	9	27	15	6	2	29	8	15	5	31	9

The number of rounds r depends on the digest size: for a d-bit digest, MD6 makes $40 + d/4$ rounds.

B Details of the Key Recovery Attack on 14-Round MD6

Table 5. Examples of maxterm equations for 14-round MD6, with respect specified cube are listed.

Maxterm equation	Output index
$A_{15}^{0} + A_{15}^{1} + A_{15}^{3} + A_{15}^{4} + A_{15}^{6} + A_{15}^{8} + A_{15}^{9} + A_{15}^{14} + A_{15}^{20} + A_{15}^{21}$ $+A_{15}^{22} + A_{15}^{26} + A_{15}^{28} + A_{15}^{32} + A_{15}^{37} + A_{15}^{38} + A_{15}^{40} + A_{15}^{41} + A_{15}^{43} + A_{15}^{44}$ $+A_{15}^{47} + A_{15}^{48} + A_{15}^{49} + A_{15}^{50} + A_{15}^{56} + A_{15}^{58} + A_{15}^{60} + A_{15}^{61} + A_{15}^{62} + A_{15}^{63}$ $+A_{16}^{1} + A_{16}^{2} + A_{16}^{3} + A_{16}^{4} + A_{16}^{5} + A_{16}^{10} + A_{16}^{11} + A_{15}^{12} + A_{16}^{13} + A_{16}^{15}$ $+A_{16}^{16} + A_{16}^{17} + A_{16}^{19} + A_{16}^{21} + A_{16}^{22} + A_{16}^{24} + A_{16}^{25} + A_{16}^{27} + A_{16}^{28} + A_{16}^{29}$ $+A_{16}^{31} + A_{16}^{32} + A_{16}^{36} + A_{16}^{37} + A_{16}^{38} + A_{16}^{39} + A_{16}^{43} + A_{16}^{44} + A_{16}^{48} + A_{16}^{49}$ $+A_{16}^{50} + A_{16}^{52} + A_{16}^{53} + A_{16}^{55} + A_{16}^{57} + A_{16}^{60} + A_{16}^{61} + A_{16}^{63} + A_{16}^{8} + 1$	O_0^0
$A_{15}^{0} + A_{15}^{1} + A_{15}^{3} + A_{15}^{6} + A_{15}^{8} + A_{15}^{10} + A_{15}^{11} + A_{15}^{14} + A_{15}^{16} + A_{15}^{21}$ $+A_{15}^{22} + A_{15}^{27} + A_{15}^{28} + A_{15}^{32} + A_{15}^{34} + A_{15}^{35} + A_{15}^{36} + A_{15}^{37} + A_{15}^{44} + A_{15}^{45}$ $+A_{15}^{48} + A_{15}^{50} + A_{15}^{54} + A_{15}^{55} + A_{15}^{57} + A_{15}^{58} + A_{15}^{59} + A_{15}^{60} + A_{15}^{63} + A_{16}^{0}$ $+A_{16}^{2} + A_{16}^{5} + A_{16}^{6} + A_{16}^{7} + A_{16}^{9} + A_{16}^{10} + A_{16}^{11} + A_{16}^{13} + A_{16}^{16} + A_{16}^{17}$ $+A_{16}^{18} + A_{16}^{19} + A_{16}^{20} + A_{16}^{21} + A_{16}^{23} + A_{16}^{30} + A_{16}^{35} + A_{16}^{36} + A_{16}^{39} + A_{16}^{42}$ $+A_{16}^{43} + A_{16}^{44} + A_{16}^{47} + A_{16}^{48} + A_{16}^{49} + A_{16}^{50} + A_{16}^{51} + A_{16}^{53} + A_{16}^{59} + A_{16}^{61}$ $+A_{16}^{50} + A_{16}^{51} + A_{16}^{53} + A_{16}^{59} + A_{16}^{61} + 1$	O_0^1