Narrow T-functions

Magnus Daum*

CITS Research Group, Ruhr-University Bochum, daum@cits.rub.de

Abstract. *T-functions* were introduced by Klimov and Shamir in a series of papers during the last few years. They are of great interest for cryptography as they may provide some new building blocks which can be used to construct efficient and secure schemes, for example block ciphers, stream ciphers or hash functions.
In the present paper, we define the *narrowness* of a T-function and study how this property affects the strength of a T-function as a cryptographic primitive. We define a new data strucure, called a *solution graph*, that enables solving systems of equations given by T-functions. The efficiency of the algorithms which we propose for solution graphs depends significantly on the narrowness of the involved T-functions. Thus the subclass of T-functions with small narrowness appears to be weak and should be avoided in cryptographic schemes.
Furthermore, we present some extensions to the methods of using solution graphs, which make it possible to apply these algorithms also to more general systems of equations, which may appear, for example, in the cryptanalysis of hash functions.

Keywords: Cryptanalysis, hash functions, solution graph, T-functions, w-narrow

1 Introduction

Many cryptanalytical problems can be described by a system of equations. A well-known example are the algebraic attacks on block and stream ciphers which use systems of multivariate quadratic equations for describing the ciphers.

However, many cryptographic algorithms use a mixture of different kinds of operations (e.g. bitwise defined functions, modular additions or multiplications and bit shifts or rotations) such that they cannot be described easily by some relatively small or simple system of linear or quadratic equations. As these operations are algebraically rather incompatible, it is hard to solve equations which include different ones algebraically.

In a series of papers [6–8] Klimov and Shamir introduced the notion of *T-functions*, in order to be able to prove theoretical results at least for some of the

* The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

constructions mentioned above. Roughly spoken, a T-function is a function for which the k-th bit of the output depends only on the first k input bits. Many basic operations available on modern microprocessors are T-functions and this means that many T-functions can be implemented very efficiently. Furthermore many of the operations mentioned above are T-functions or very similar to T-functions.

In this paper we concentrate on a certain subclass of T-functions, which we call *w-narrow T-functions*. In a w-narrow T-function the dependance of the k-th output bit on the first k input bits is even more restricted: The k-th output bit must be computable from only the k-th input bits and some information of a length of w bits computed from the first $k-1$ input bits.

We present a data structure, called a *solution graph*, which allows to efficiently represent the set of solutions of an equation, which can be described by a w-narrow T-function. The smaller w is, the more efficient is this representation. Additionally we present a couple of algorithms which can be used for analysing and solving such systems of equations described by T-functions. These algorithms include enumerating all solutions, computing the number of solutions, choosing random solutions and also combining two or more solution graphs, e.g. to compute the intersection of two sets of solutions or to compute the concatenation of two T-functions.

However, this paper is not only dedicated to the quite young subject of T-functions. The solution graphs together with the presented algorithms, can be used for cryptanalysis in a lot of contexts, for example also in the cryptanalysis of hash functions. In his attacks on the hash functions MD4, MD5 and RIPEMD (see [3–5]), Dobbertin used, as one key ingredient, an algorithm which can be described as some kind of predecessor of the algorithms used for constructing solution graphs and enumerating all the solutions (see Appendix A). In this paper we also describe some extensions which allow to apply the algorithms also in contexts which are a little more general than systems of equations describable by "pure" T-functions.

We start in Section 2 by defining the narrowness of a T-function and give some basic examples and properties. Then in Section 3 we describe the new data structure, the solution graph, and give an algorithm for constructing solution graphs from systems of equations of T-functions. Section 4 gives further algorithms for solution graphs.
In Section 5 we present some possible extensions to the definition of a solution graph, which allow to apply these algorithms also in more general situations, for example in the cryptanalysis of hash functions

In Appendix A we describe the ideas and the original algorithm used by Dobbertin in his attacks. Two actual examples of systems coming from the cryptanalysis of hash functions, which have been solved successfully with solution graphs are given in Appendix B.

In this extended abstract the proofs are omitted. They can be found in [1] and the full version [2] of this paper.

2 Notation and Definitions

For the convenience of the reader, we mainly adopt the notation of [9]. Especially, let n be the word size in bits, $\mathbb{B}$ be the set $\{0,1\}$ and let $[x]_i$ denote the i-th bit of the word $x \in \mathbb{B}^n$, where $[x]_0$ is the least significant bit of x. Hence, $x = ([x]_{n-1}, \dots, [x]_0)$ also stands for the integer $\sum_{i=0}^{n-1} [x]_i 2^i$.

If $x = (x_0, \dots, x_{m-1})^T \in \mathbb{B}^{m \times n}$ is a column vector of m words of n bits, then $[x]_i$ stands for the column vector $([x_0]_i, \dots, [x_{m-1}]_i)^T$ of the i-th bits of those words.

By $x \ll s$ we will denote a left shift by s positions and by $x \lll r$ we denote a left rotation (a cyclic shift) by r positions.

Let us first recall the definition of a T-function from [9]:

Definition 1 (T-Function). *A function $f : \mathbb{B}^{m \times n} \to \mathbb{B}^{l \times n}$ is called a* T-function *if the k-th column of the output $[f(x)]_{k-1}$ depends only on the first k columns of the input $[x]_{k-1}, \dots, [x]_0$:*

$$\begin{pmatrix} [x]_0 \\ [x]_1 \\ [x]_2 \\ \vdots \\ [x]_{n-1} \end{pmatrix}^T \mapsto \begin{pmatrix} f_0([x]_0) \\ f_1([x]_0, [x]_1) \\ f_2([x]_0, [x]_1, [x]_2) \\ \vdots \\ f_{n-1}([x]_0, [x]_1, \dots, [x]_{n-1}) \end{pmatrix}^T \tag{1}$$

There are many examples for T-functions. All bitwise defined functions, e.g. a Boolean operation like $(x, y) \mapsto x \wedge y$ or the majority function $(x, y, z) \mapsto (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$, are T-functions, because the k-th output bit depends only on the k-th input bits. But also other common functions, like addition or multiplication of integers (modulo 2^n) are T-functions, as can be easily seen from the schoolbook methods. For example, when executing an addition, to compute the k-th bit of the sum, the only necessary information (besides the k-th bits of the addends) is the carrybit coming from computing the $(k-1)$-th bit.

This is also a good example for some other more special property that many T-functions have: You need much less information than "allowed" by the definition of a T-function: In order to compute the k-th output column $[f(x)]_{k-1}$ you need only the k-th input column $[x]_{k-1}$ and very little information about the first $k-1$ columns $[x]_{k-2}, \dots, [x]_0$, for example some value $\alpha_k([x]_{k-2}, \dots, [x]_0) \in \mathbb{B}^w$ of w bits width. This leads to our definition of a *w-narrow T-function*:

Definition 2 (w-narrow).
A T-function f is called w-narrow *if there are mappings*

$$\alpha_1 : \mathbb{B}^m \to \mathbb{B}^w, \qquad \alpha_k : \mathbb{B}^{m+w} \to \mathbb{B}^w, k = 2, \dots, n-1 \tag{2}$$

and auxiliary variables

$$a_1 := \alpha_1([x]_0), \qquad a_k := \alpha_k([x]_{k-1}, a_{k-1}), k = 2, \dots, n-1 \tag{3}$$

such that f can be written as

$$\begin{pmatrix} [x]_0 \\ [x]_1 \\ [x]_2 \\ [x]_3 \\ \vdots \\ [x]_{n-1} \end{pmatrix}^T \mapsto \begin{pmatrix} f_0([x]_0) \\ f_1([x]_1\,, a_1) \\ f_2([x]_2\,, a_2) \\ f_3([x]_3\,, a_3) \\ \vdots \\ f_{n-1}([x]_{n-1}\,, a_{n-1}) \end{pmatrix}^T \tag{4}$$

The smallest w such that some f is w-narrow is called the narrowness *of f.*

Let us take a look at some examples of w-narrow T-functions.

Example 1.

1. The identity function and all bitwise defined functions are 0-narrow.
2. As described above, addition of two integers modulo 2^n is a 1-narrow T-function, as you only need to remember the carrybit in each step.
3. A left shift by s bits is an s-narrow T-function.
4. Each T-function $f : \mathbb{B}^{m\times n} \to \mathbb{B}^{l\times n}$ is $(m(n-1))$-narrow.

Directly from Definition 2 one can derive the following lemma about the composition of narrow functions:

Lemma 1. *Let $f, g_1, \ldots, g_r$ be T-functions which are w_f-, w_{g_1}-,...,w_{g_r}-narrow respectively. Then the function h defined by*

$$h(x) := f(g_1(x), \ldots, g_r(x))$$

is $(w_f + w_{g_1} + \ldots + w_{g_r})$-narrow.

Note that this lemma (as the notion of w-narrow itself) gives only an upper bound on the narrowness of a function: For example, the addition of 4 integers can be composed of three (1-narrow) 2-integer-additions. Thus by Lemma 1 it is 3-narrow. But it is also 2-narrow, because the carry value to remember can never become greater than 3 (which can be represented in $\mathbb{B}^2$) when adding 4 bits and a maximum (earlier) carry of 3.

3 Solution Graphs for Narrow T-functions

In this section we will describe a data structure which allows to represent the set of solutions of a system of equations of T-functions.
Common approaches for finding solutions of such equations are doing an exhaustive or randomized search or using some more sophisticated algorithms as the one used by Dobbertin in his attacks on the hash functions MD4, MD5 and RIPEMD in [3–5]. This algorithm, which gave us the idea of introducing the data structure presented here, is described in Appendix A.

In general, the trees build in Dobbertin's algorithm and thus its complexity, needed for building them, may become quite large, in the worst case up to the complexity of an exhaustive search. But this can be improved a lot in many cases, or, to be more precise, in the case of T-functions which are w-narrow for some small w, as we will show in the sequel.

Let us first note, that it suffices to consider only the problem of solving one equation

$$f(x) = 0, \tag{5}$$

where $f : \mathbb{B}^{m\times n} \to \mathbb{B}^n$ is some T-function:
If we had an equation described by two T-functions $g(x) = h(x)$ we could simply define $\hat{g}(x) := g(x) \oplus h(x)$ and consider the equation $\hat{g}(x) = 0$ instead. If we had a system of several such equations $\hat{g}_1(x) = 0, \ldots, \hat{g}_r(x) = 0$ (or a function $\hat{g} : \mathbb{B}^{m\times n} \to \mathbb{B}^{l\times n}$ with component functions $\hat{g}_1, \ldots, \hat{g}_r$) we could simply define $f(x) := \bigvee_{i=1}^{r} \hat{g}_i(x)$ and consider only the equation $f(x) = 0$.
As both operations, $\oplus$ and $\vee$, are 0-narrow, due to Lemma 1, the narrowness of f is at most the sum of the narrownesses of the involved functions.

If f in (5) is a w-narrow T-function for some "small" w, a solution graph, as given in the following definition, can be efficiently constructed and allows many algorithms which are useful for cryptanalysing such functions.

Definition 3 (Solution Graph). *A directed graph $\mathcal{G}$ is called a* solution graph *for an equation $f(x) = 0$ where $f : \mathbb{B}^{m\times n} \to \mathbb{B}^n$, if the following properties hold:*

1. *The vertices of $\mathcal{G}$ can be arranged in $n+1$ layers such that each edge goes from a vertex in layer l to some vertex in layer $l+1$ for some $l \in \{0, \ldots, n-1\}$.*
2. *There is only one vertex in layer 0, called the* root*.*
3. *There is only one vertex in layer n, called the* sink*.*
4. *The edges are labelled with values from $\mathbb{B}^m$ such that the labels for all edges starting in one vertex are pairwise distinct.*
5. *There is a 1-to-1 correspondence between paths from the root to the sink in $\mathcal{G}$ and solutions of the equation $f(x) = 0$:*
   For each solution x there exists a path from the root to the sink such that the k-th edge on this path is labelled with $[x]_{k-1}$ and vice versa.

The maximum number of vertices in one layer of a solution graph $\mathcal{G}$ is called the width *of $\mathcal{G}$.*

In the following we will describe how to efficiently construct a solution graph which represents the complete set of solutions of (5). Therefore let f be w-narrow with some auxiliary functions $\alpha_1, \ldots, \alpha_{n-1}$ as in Definition 2. To identify the vertices during the construction we label them with two numbers (l, a) each, where $l \in \{0, \ldots, n\}$ is the number of the layer and $a \in \mathbb{B}^w$ corresponds to a possible output of one of the auxiliary functions α_i. This labelling is only required for the construction and can be deleted afterwards.

Then the solution graph can be constructed by the following algorithm:

Algorithm 1 (Construction of a Solution Graph).

1. Start with one vertex labelled with $(0, *)$
2. For each possible value for $[x]_0$, for which it holds that $f_0([x]_0) = 0$:
   Add an edge
   $$(0, *) \longrightarrow (1, \alpha_1([x]_0))$$
   and label this edge with the value of $[x]_0$.
3. For each layer l, $l \in \{1, \ldots, n-2\}$, and each vertex (l, a_l) in layer l:
   For each possible value for $[x]_l$ for which $f_l([x]_l, a_l) = 0$:
   Add some edge
   $$(l, a_l) \longrightarrow (l+1, \alpha_{l+1}([x]_l, a_l))$$
   and label this edge with the value of $[x]_l$.
4. For each vertex $(n-1, a)$ in layer $n-1$ and each possible value for $[x]_{n-1}$ for which $f_{n-1}([x]_{n-1}, a) = 0$:
   Add an edge
   $$(n-1, a) \longrightarrow (n, *)$$
   and label it with the value of $[x]_{n-1}$.

Toy examples of the results of this construction can be found in Figure 1. Compared with the trees in Figure 5 and 6, resulting from Dobbertin's algorithm, this shows that these solution graphs are much more efficient.

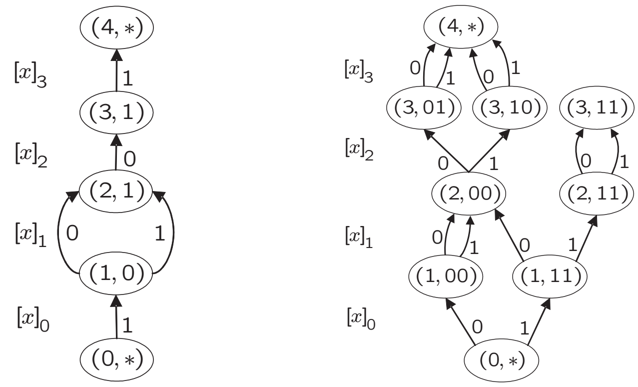

Fig. 1. Solution graphs for the equations $((x \vee 0010_2) + 0110_2) \oplus 0001_2 = 0$ (on the left) and $((0100_2 \oplus (x + 0101_2)) - (0100_2) \oplus x) \oplus 1101_2 = 0$ (on the right) with $n = 4$.

From the description of Algorithm 1 the following properties can be easily deduced:

Theorem 1. *Let $f : \mathbb{B}^{m \times n} \to \mathbb{B}^n$ be a w-narrow T-function and $\mathcal{G}$ the graph for $f(x) = 0$ constructed by Algorithm 1. Then $\mathcal{G}$*

– *is a solution graph for* $f(x) = 0$,
– *has width at most* 2^w, *i.e.* $\mathcal{G}$ *has* $v \leq (n-1)2^w + 2$ *vertices and* $e \leq (v-1)2^m$ *edges.*

Proof. For the proof, see [1] or the full version [2] of this paper.

This theorem gives an upper bound on the size of the constructed solution graph, which depends significantly on the narrowness of the examined function f. This shows that, as long as f is w-narrow for some small w, such a solution graph can be constructed quite efficiently.

4 Algorithms for Solution Graphs

The design of a solution graph, as presented in Section 3 is very similar to that of binary decision diagrams (BDDs). Thus it is not surprising, that many ideas of algorithms for BDDs can be adopted to construct efficient algorithms for solution graphs. For an introduction to the subject of BDDs, see for example [10].

The complexity of these algorithms naturally depends mainly on the size of the involved solution graphs. Thus, we will first describe how to reduce this size.

4.1 Reducing the Size

We describe this using the example of the solution graph on the right hand side of Figure 1: There are no edges starting in $(3, 11)$ and thus there is no path from the root to the sink which crosses this vertex. This means, due to Definition 3, this vertex is of no use for representing any solution, and therefore it can be deleted. After this deletion the same applies for $(2, 11)$ and thus this vertex can also be deleted.

For further reduction of the size let us define what we mean by *equivalent* vertices:

Definition 4. *Two vertices* a *and* b *in a solution graph are called* equivalent*, if for each edge* $a \to c$ *(with some arbitrary vertex* c*) labelled with* x *there is an edge* $b \to c$ *labelled with* x *and vice versa.*

For the reduction of the size, it is important to notice the following lemma:

Lemma 2. *If* a *and* b *are equivalent, then there are the same paths (according to the labelling of their edges) from* a *to the sink as from from* b *to the sink.*

For example let us now consider the vertices $(3, 01)$ and $(3, 10)$. From each of these two vertices there are two edges, labelled with 0 and 1 respectively, which point to $(4, *)$ and thus these two vertices are equivalent. According to Lemma 2 this means that a path from the root to one of those two vertices can be extended to a path to the sink by the same subpaths, independently of whether it goes through $(3, 01)$ or $(3, 10)$. Due to the defining property of a solution graph, this means, that we can merge these two equivalent vertices into one, reducing the

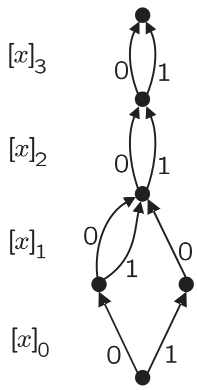

Fig. 2. Solution graph for the equation $((\mathtt{0100}_2 \oplus (x + \mathtt{0101}_2)) - (\mathtt{0100}_2 \oplus x)) \oplus \mathtt{1101}_2 = 0$ (compare Figure 1) after reducing its size.

size once more. The resulting solution graph is presented in Figure 2. In this figure the labels of the vertices are omitted as they are only required for the construction algorithm.

Of course, merging two equivalent vertices, and also the deletion of vertices as described above, may again cause two vertices to become equivalent, which have not been equivalent before. But this concerns only vertices in the layer below the layer in which two vertices were merged. Thus for the reduction algorithm it is important to work from top (layer $n-1$) to bottom (layer 1):

Algorithm 2 (Reduction of the Size).

1. Delete each vertex (together with corresponding edges) for which there is no path from the root to this vertex or no path from this vertex to the sink.
2. For each layer l starting from $n-1$ down to 1 merge all pairs of vertices in layer l which are equivalent.

To avoid having to check all possible pairs of vertices in one layer for equivalence separately to find the equivalent vertices (which would result in a quadratic complexity), in Algorithm 2 one should first sort the vertices of the active layer according to their set of outgoing edges. Then equivalent vertices can be found in linear time.

Similar to what can be proven for ordered BDDs, for solution graphs reduced by Algorithm 2 it can be shown that they have minimal size:

Theorem 2. *Let $\mathcal{G}$ be a solution graph for some function f and let $\tilde{\mathcal{G}}$ be the output of Algorithm 2 applied to $\mathcal{G}$. Then there is no solution graph for f which has less vertices than $\tilde{\mathcal{G}}$.*

Proof. For the proof, see [1] or the full version [2] of this paper. □

With the help of the following theorem it is possible to compute the narrowness of f, i.e. the smallest value w such that f is w-narrow. Like Theorem 1 gives a bound on the width of a solution graph based on a bound for the narrowness of the considered function, the following theorem provides the other direction:

Theorem 3. *Let $f : \mathbb{B}^{m\times n} \to \mathbb{B}^n$ be a T-function and define $\tilde{f} : \mathbb{B}^{(m+1)\times n} \to \mathbb{B}^n$ by $\tilde{f}(x, y) := f(x) \oplus y$.*
If $\mathcal{G}$ is a minimal solution graph of width W for the equation $\tilde{f}(x, y) = 0$, then f is a $\lceil \log_2 W \rceil$-narrow T-function.

Proof. For the proof, see [1] or the full version [2] of this paper. □

In the following we always suppose that we have solution graphs of minimal size (from Algorithm 2 and Lemma 2) as inputs.

4.2 Computing Solutions

Similar to what can be done by Dobbertin's algorithm (see Algorithm 7 in Appendix A), a solution graph can also be used to enumerate all the solutions:

Algorithm 3 (Enumerate Solutions).
Compute all possible paths from the root to the sink by a depth-first search and output the corresponding labelling of the edges.

Of course, the complexity of this algorithm is directly related to the number of solutions. If there are many solutions, it is similar to the complexity of an exhaustive search (as for Algorithm 7), simply because all of them need to be written. But if there are only a few, it is very fast, usually much faster than Algorithm 7.

However, often we are only interested in the number of solutions of an equation which can be computed much more efficiently, namely, with a complexity linear in the size of the solution graph. The following algorithm achieves this by labeling every vertex with the number of possible paths from that vertex to the sink. Then the number computed for the root gives the number of solutions:

Algorithm 4 (Number of Solutions).

1. Label the sink with 1.
2. For each layer l from $n - 1$ down to 0:
   Label each vertex A in l with the sum of the labels of all vertices B (in layer $l + 1$) for which an edge $A \to B$ exists.
3. Output the label of the root.

An application of this algorithm is illustrated in Figure 3.

After having labelled all vertices by Algorithm 4 it is even possible to choose solutions from the represented set uniformly at random:

Algorithm 5 (Random Solution).
Prerequisite: The vertices have to be labelled as in Algorithm 4.

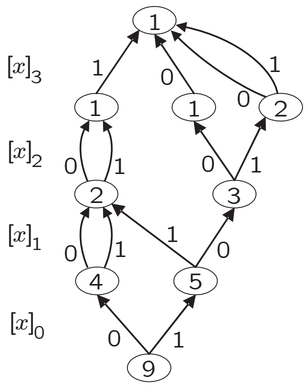

Fig. 3. A solution graph after application of Algorithm 4.

1. Start at the root.
2. Repeat
   – From the active vertex A (labelled with n_A) randomly choose one outgoing edge such that the probability that you choose $A \to B$ is $\frac{n_B}{n_A}$ where n_B is the label of B.

   – Remember the label of $A \to B$
   – Make B the active vertex.

   until you reach the sink.
3. Output the solutions corresponding to the remembered labels of the edges on the chosen path.

4.3 Combining Solution Graphs

So far, we only considered the situation in which the whole system of equations is reduced to one equation $f(x) = 0$, as described at the beginning of Section 3, and then a solution graph is constructed from this equation. Sometimes it is more convenient to consider several (systems of) equations separately and then combine their sets of solutions in some way. Therefore let us now consider two equations

$$g(x_1, \dots, x_r, y_1, \dots, y_s) = 0 \tag{6}$$
$$h(x_1, \dots, x_r, z_1, \dots, z_t) = 0 \tag{7}$$

which include some common variables $x_1, \dots, x_r$ as well as some distinct variables $y_1, \dots, y_s$ and $z_1, \dots, z_t$ respectively. Let $\mathcal{G}_g$ and $\mathcal{G}_h$ be the solution graphs for (6) and (7) respectively.

Then the set of solutions of the form $(x_1, \dots, x_r, y_1, \dots, y_s, z_1, \dots, z_t)$ which fulfill both equations simultaneously can be computed by the following algorithm.

Algorithm 6 (Intersection). Let the vertices in $\mathcal{G}_g$ be labelled with $(l, a_g)_g$ where l is the layer and a_g is some identifier which is unique per layer, and those of $\mathcal{G}_h$ analogously with some $(l, a_h)_h$. Then construct a graph whose vertices will be labelled with (l, a_g, a_h) by the following rules:

1. Start with the root $(0, *_g, *_h)$.
2. For each layer $l \in \{0, \ldots, n-1\}$ and each vertex (l, a_g, a_h) in layer l:
   – Consider each pair of edges

$$((l, a_g)_g \to (l+1, b_g)_g, (l, a_h)_h \to (l+1, b_h)_h)$$

   labelled with

$$(X_g, Y_g) = ([x_1]_l, \ldots, [x_r]_l, [y_1]_l, \ldots, [y_s]_l)$$
$$\text{and } (X_h, Z_h) = ([x_1]_l, \ldots, [x_r]_l, [z_1]_l, \ldots, [z_t]_l) \text{ respectively.}$$

   – If $X_g = X_h$, add an edge

$$(l, a_g, a_h) \to (l+1, b_g, b_h)$$

   and label it with (X_g, Y_g, Z_h).

The idea of this algorithm is to traverse the two input graphs $\mathcal{G}_g$ and $\mathcal{G}_h$ in parallel and to simulate computing both functions in parallel in the output graph by storing all necessary information in the labels of the output graph. For an illustration of this algorithm, see Figure 4. Also notice that this algorithm can be easily generalized to having more than two input graphs.

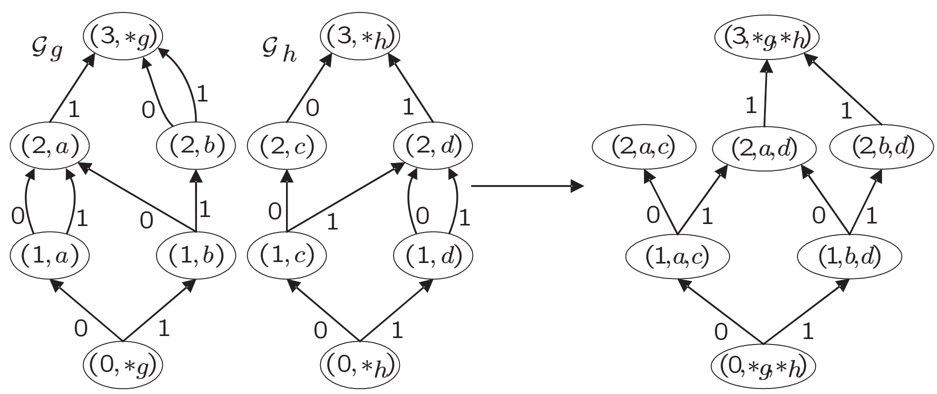

Fig. 4. Intersection of two solution graphs by Algorithm 6.

Apart from just computing mere intersections of sets of solutions, Algorithm 6 can also be used to solve equations given by the concatenation of two T-functions:

$$f(g(x)) = y \tag{8}$$

To solve this problem, just introduce some auxiliary variable z and apply Algorithm 6 to the two solution graphs which can be constructed for the equations $f(z) = y$ and $g(x) = z$ respectively.

Combining this idea (applied to the situation $f = g$) with some square-and-multiply technique, allows for some quite efficient construction of a solution graph for an equation of the form $f^i(x) = y$ with some (small) fixed value i. This may be of interest for example for cryptanalysing stream ciphers which are constructed as suggested for example by Klimov in [9], but use T-functions with some small narrowness instead of one of the functions proposed by Klimov which seem to have a large narrowness.

5 Extensions of this Method

In many cryptographical systems the operations used are usually *not* restricted to T-functions. Often such systems also include other basic operations, as, for example, right bit shifts or bit rotations, which are quite similar, but not T-functions according to Definition 1. Hence, systems of equations used in the cryptanalysis of such ciphers usually cannot be solved directly by applying solution graphs as presented in Sections 3 and 4. In this section we give some examples of how such situations can be handled, for example by extending the definition of a solution graph such that it is still applicable.

5.1 Including Right Shifts

Let us first consider a system of equations which includes only T-functions and some right shift expressions $x \gg r$. This can be transformed by substituting every appearance of $x \gg r$ by an auxiliary variable z_r and adding an extra equation

$$z_r \ll r = x \wedge (\underbrace{11\ldots1}_{n-r}\underbrace{0\ldots0}_{r}) \tag{9}$$

which defines the relationship between x and z_r. Then the resulting system is completely described by T-functions and can be solved with a solution graph.

Here, similarly as when solving (8) some problem occurs: We have to add an extra (auxiliary) variable z, which potentially increases the size of the needed solution graph. This is even worse as the solution graph stores all possible values of z corresponding to solutions for the other variables, even if we are not interested in them at all. This can be dealt with by softening Definition 3 to *generalized solutions graphs:*

5.2 Generalized Solution Graphs

For a generalized solution graph we require every property from Definition 3 with the exception that the labels of edges starting in one vertex are *not* required to be pairwise distinct.

Then we can use similar algorithms as those described above, e.g. for reducing the size or combining two graphs. But usually these algorithms are a little bit more sophisticated: For example, for minimizing the size, it does not suffice to consider equivalent vertices as defined in Definition 4. In a generalized solution graph it is also possible that the sets of *incoming* edges are equal and, clearly, two such vertices with equal sets of incoming edges (which we will also call equivalent in the case of general solution graphs) can also be merged. But this also means that merging two equivalent vertices in layer l may not only cause vertices in layer $l-1$ to become equivalent, but also vertices in layer $l+1$. Thus, in the generalized version of Algorithm 2 we have to go back and forth in the layers to ensure that in the end there are no equivalent vertices left.

This definition of a generalized solution graph allows to "remove" variables without losing the information about their existence. This means, instead of representing the set $\{(x,y) \mid f(x,y) = 0\}$ with a solution graph $\mathcal{G}$, we can represent the set $\{x \mid \exists y : f(x,y) = 0\}$ with a solution graph $\mathcal{G}'$ which is constructed from $\mathcal{G}$ by simply deleting the parts of the labels which correspond to y. Of course, this does not decrease the size of the generalized solution graph directly but (hopefully) it allows further reductions of the size.

5.3 Including Bit Rotations

Let us now take a look at another commonly used function which is not a T-function, a bit rotation by r bits:

$$f(x) := x \lll r \tag{10}$$

If we would fix the r most significant bits of x, for example to some value c, then this function can be described by a bit shift of r positions and a bitwise defined function

$$f(x) := (x \ll r) \vee c \tag{11}$$

which is an r-narrow T-function. Thus, by looping over all 2^r possible values for c an equation involving (10) can also be solved by solution graphs.

If we use generalized solution graphs, it is actually possible to combine all 2^r such solution graphs to one graph, in which again the complete set of solutions is represented: This can be done by simply merging all the roots and all the sinks of the 2^r solution graphs as they are clearly equivalent in the generalized sense.

Two examples of actual systems of equations which were solved by applying solution graphs and the extensions from this section are given in Appendix B.

6 Conclusion

In this paper we defined a subclass of weak T-functions, the *w-narrow T-functions*. We showed that systems of equations involving only w-narrow T-functions (with small w) can be solved efficiently by using *solution graphs* and thus such functions should be avoided in cryptographical schemes.

Let us stress again that this does *not* mean that the concept of using T-functions for constructing cryptosystems is bad. One just has to assure that the used T-functions are not too narrow. For example, it is a good idea to always include multiplications and bit shifts of some medium size in the functions, as those are examples of T-functions which are not very narrow.

Additionally we presented some extensions to our proposal of a solution graph. These extensions allow to use the solution graphs also in other contexts than pure T-functions, for example as a tool in the cryptanalysis of hash functions.

Acknowledgements. This work is part of PhD thesis [1] and I would like to thank my supervisor Hans Dobbertin for support and discussions. Also I would like to thank Tanja Lange for many helpful discussions and comments.

References

1. M. Daum. *Cryptanalysis of Hash Functions of the MD4-Family*. PhD Thesis, Ruhr-University Bochum, in preparation.
2. M. Daum. *Narrow T-functions*. Cryptology ePrint Archive, Report 2005/016. (available under: http://eprint.iacr.org/2005/016)
3. H. Dobbertin: *The status of MD5 after a recent attack.* CryptoBytes, 2(2), 1996, pp. 1-6.
4. H. Dobbertin: *RIPEMD with two-round compress function is not collision-free.* Journal of Cryptology 10, 1997, pp. 51-68.
5. H. Dobbertin: *Cryptanalysis of MD4.* Journal of Cryptology 11, 1998, pp. 253-274.
6. A. Klimov and A. Shamir: *A New Class of Invertible Mappings.* Workshop on Cryptographic Hardware and Embedded Systems (CHES), 2002.
7. A. Klimov and A. Shamir: *Cryptographic Applications of T-functions.* Selected Areas in Cryptography (SAC), 2003.
8. A. Klimov and A. Shamir: *New Cryptographic Primitives Based on Multiword T-functions.* FSE 2004, 2004.
9. A. Klimov: *Applications of T-functions in Cryptography.* PhD Thesis, Weizmann Institute of Science, submitted, 2004.
(available under: http://www.wisdom.weizmann.ac.il/˜ask/)
10. I. Wegener: *Branching Programs and Binary Decision Diagrams: Theory and Applications.* SIAM Monographs on Discrete Mathematics and Applications, 2000.

A Dobbertin's Original Algorithm from the Attacks on MD4, MD5 and RIPEMD

In this section we describe the algorithm used by Dobbertin in his attacks from [3–5]. However, we do this using the same terminology as in the other sections of the present paper to maximize the comparability.

Let $\mathcal{S}$ be a system of equations which can be completely described by T-functions and let $\mathcal{S}_k$ denote the system of equations in which only the k least significant bits of each equation are considered. As those k bits only depend on the k least significant bits of all the inputs, we will consider the solutions of $\mathcal{S}_k$ to have only k bits per variable as well.

Then, from the defining property of a T-function, the following theorem easily follows:

Theorem 4. *Every solution of $\mathcal{S}_k$ is an extension of a solution of $\mathcal{S}_{k-1}$.*

This theorem directly leads to the following algorithm for enumerating all the solutions of $\mathcal{S}$.

Algorithm 7.

1. Find all solutions (having only 1 bit per variable) of $\mathcal{S}_1$.
2. For every found solution of some $\mathcal{S}_k, k \in \{1, \ldots, n-1\}$, recursively check which extensions of this solution by 1 bit per variable are solutions of $\mathcal{S}_{k+1}$.
3. Output the found solutions of $\mathcal{S}_n (= \mathcal{S})$.

An actual toy example application of this algorithm – finding the solutions x of the equation $\mathcal{S}$ given by $(x \vee \mathtt{0010}_2) + \mathtt{0110}_2 = \mathtt{0001}_2$ with $n = 4$ – is illustrated in Figure 5: We start at the root of the tree and check whether 0 or 1 are possible values for $[x]_0$, i.e. if they are solutions of $\mathcal{S}_1$ which is given by $([x]_0 \vee 0) + 0 = 1$. Obviously 0 is not a solution of this equation and thus we need not consider any more values for x starting with 0. But 1 is a solution of $\mathcal{S}_1$, thus we have to check whether extensions (i.e. $\mathtt{01}_2$ or $\mathtt{11}_2$) are solutions of $\mathcal{S}_2$: $(x \vee \mathtt{10}_2) + \mathtt{10}_2 = \mathtt{01}_2$. Doing this recursively finally leads to the "tree of solutions", illustrated on the left hand side of Figure 5.

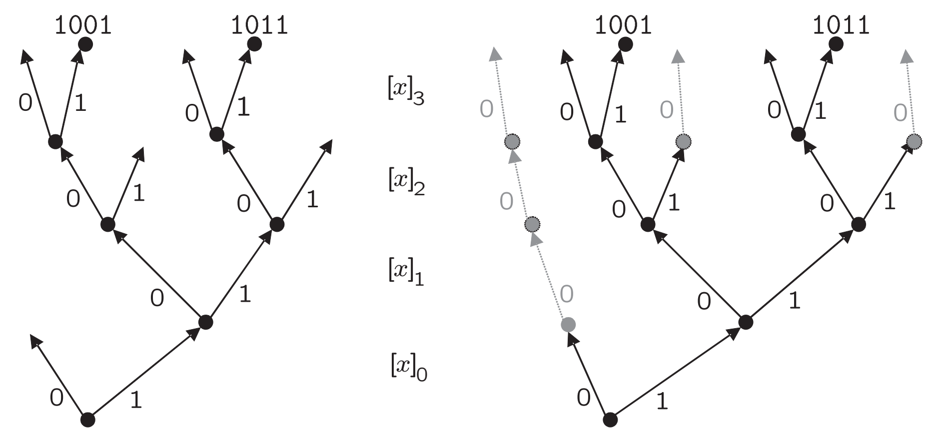

Fig. 5. "Solution tree" for the equation $(x \vee \mathtt{0010}_2) + \mathtt{0110}_2 = \mathtt{0001}_2$ with $n = 4$.

If this method is implemented directly as described in Algorithm 7, it has a worst case complexity which is about twice as large as that of an exhaustive

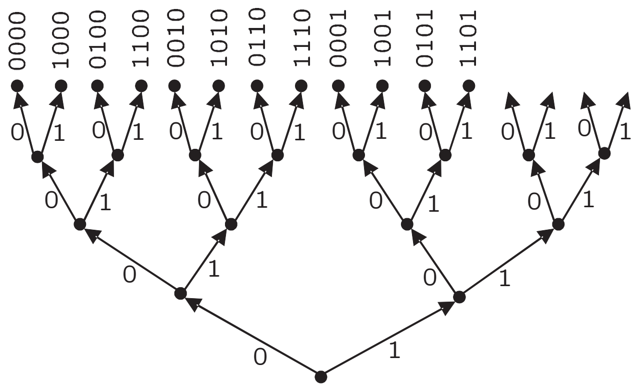

Fig. 6. "Solution tree" for the equation $(\texttt{0100}_2 \oplus (x+\texttt{0101}_2)) - (\texttt{0100}_2 \oplus x) = \texttt{1101}_2$ with $n = 4$.

search, because the full solution tree of depth n has $2^{n+1}-1$ vertices. An example of such a "worst case solution tree" is given in Figure 6. To actually achieve a worst case complexity similar to that of an exhaustive search a little modification is necessary to the algorithm: The checking should be done for complete paths (as indicated by the *grey* arrows in the tree on the right hand side in Figure 5), which can also be done in one machine operation, and not bit by bit. This means, we would start by checking $\texttt{0000}_2$ and recognize that this fails already in the least significant bit. In the next step we would check $\texttt{0001}_2$ and see that the three least significant bits are okay. This means in the following step we would only change the fourth bit and test $\texttt{1001}_2$ which would give us the first solution. All in all we would need only 7 checks for this example as indicated by the grey arrows.

The worst case complexity of this modified algorithm (which is what was actually implemented in Dobbertin's attacks) is clearly 2^n as this is the number of leaves of a full solution tree. However, it is also quite clear, that in the average case, or rather in the case of fewer solutions, this algorithm is much more efficient.

B Examples of Applications

In this section we present two examples of systems of equations which were actually solved by using the techniques presented in this paper. They have both appeared as one small part in an attempt to apply Dobbertin's methods from [3–5] to SHA-1. In this paper we concentrate on describing how these systems were solved and omit a detailed description of their meanings.

The first system comes from looking for so-called "inner collisions" and includes 14 equations and essentially 22 variables $R_1, \ldots, R_{13}, \varepsilon_3, \ldots, \varepsilon_{11}$:

$$
\begin{aligned}
0 &= \varepsilon_3 + 1 \\
0 &= \varepsilon_4 - (\widetilde{R_3} \lll 5 - R_3 \lll 5) + 1 \\
\mathrm{Ch}(\widetilde{R_3}, R_2 \lll 30, R_1 \lll 30) - \mathrm{Ch}(R_3, R_2 \lll 30, R_1 \lll 30) &= \varepsilon_5 - (\widetilde{R_4} \lll 5 - R_4 \lll 5) + 1 \\
\mathrm{Ch}(\widetilde{R_4}, \widetilde{R_3} \lll 30, R_2 \lll 30) - \mathrm{Ch}(R_4, R_3 \lll 30, R_2 \lll 30) &= \varepsilon_6 - (\widetilde{R_5} \lll 5 - R_5 \lll 5) \\
\mathrm{Ch}(\widetilde{R_5}, \widetilde{R_4} \lll 30, \widetilde{R_3} \lll 30) - \mathrm{Ch}(R_5, R_4 \lll 30, R_3 \lll 30) &= \varepsilon_7 - (\widetilde{R_6} \lll 5 - R_6 \lll 5) + 1 \\
\mathrm{Ch}(\widetilde{R_6}, \widetilde{R_5} \lll 30, \widetilde{R_4} \lll 30) - \mathrm{Ch}(R_6, R_5 \lll 30, R_4 \lll 30) &= \varepsilon_8 - (\widetilde{R_7} \lll 5 - R_7 \lll 5) \\
&\quad - (\widetilde{R_3} \lll 30 - R_3 \lll 30) + 1 \\
\mathrm{Ch}(\widetilde{R_7}, \widetilde{R_6} \lll 30, \widetilde{R_5} \lll 30) - \mathrm{Ch}(R_7, R_6 \lll 30, R_5 \lll 30) &= \varepsilon_9 - (\widetilde{R_8} \lll 5 - R_8 \lll 5) \\
&\quad - (\widetilde{R_4} \lll 30 - R_4 \lll 30) + 1 \\
\mathrm{Ch}(\widetilde{R_8}, \widetilde{R_7} \lll 30, \widetilde{R_6} \lll 30) - \mathrm{Ch}(R_8, R_7 \lll 30, R_6 \lll 30) &= \varepsilon_{10} - (\widetilde{R_9} \lll 5 - R_9 \lll 5) \\
&\quad - (\widetilde{R_5} \lll 30 - R_5 \lll 30) \\
\mathrm{Ch}(\widetilde{R_9}, \widetilde{R_8} \lll 30, \widetilde{R_7} \lll 30) - \mathrm{Ch}(R_9, R_8 \lll 30, R_7 \lll 30) &= \varepsilon_{11} - (\widetilde{R_{10}} \lll 5 - R_{10} \lll 5) \\
&\quad - (\widetilde{R_6} \lll 30 - R_6 \lll 30) \\
\mathrm{Ch}(\widetilde{R_{10}}, \widetilde{R_9} \lll 30, \widetilde{R_8} \lll 30) - \mathrm{Ch}(R_{10}, R_{19} \lll 30, R_8 \lll 30) &= -(\widetilde{R_{11}} \lll 5 - R_{11} \lll 5) \\
&\quad - (\widetilde{R_7} \lll 30 - R_7 \lll 30) + 1 \\
\mathrm{Ch}(\widetilde{R_{11}}, \widetilde{R_{10}} \lll 30, \widetilde{R_9} \lll 30) - \mathrm{Ch}(R_{11}, R_{10} \lll 30, R_9 \lll 30) &= -(\widetilde{R_8} \lll 30 - R_8 \lll 30) \\
\mathrm{Ch}(R_{12}, \widetilde{R_{11}} \lll 30, \widetilde{R_{10}} \lll 30) - \mathrm{Ch}(R_{12}, R_{11} \lll 30, R_{10} \lll 30) &= -(\widetilde{R_9} \lll 30 - R_9 \lll 30) \\
\mathrm{Ch}(R_{13}, R_{12} \lll 30, \widetilde{R_{11}} \lll 30) - \mathrm{Ch}(R_{13}, R_{12} \lll 30, R_{11} \lll 30) &= -(\widetilde{R_{10}} \lll 30 - R_{10} \lll 30) + 1 \\
0 &= -(\widetilde{R_{11}} \lll 30 - R_{11} \lll 30) + 1
\end{aligned}
$$

Here we use $\widetilde{R_i} := R_i + \varepsilon_i$ for a compact notation, the word size is $n = 32$, and the Ch in these equations stands for the bitwise defined choose-function

$$\mathrm{Ch}(x, y, z) = (x \wedge y) \vee (\overline{x} \wedge z).$$

It was not possible to solve this system in full generality, but for the application it sufficed to find some fixed values for $\varepsilon_3, \ldots, \varepsilon_{11}$ such that there are many solutions for the R_i and then to construct a generalized solution graph for the solutions for $R_1, \ldots, R_{13}$.

The choice for good values for some of the ε_i could be done by either theoretical means or by constructing solution graphs for single equations of the system and counting solutions with fixed values for some ε_i.
For example, from the solution graph for the last equation it is possible (as described in Section 5.2) to remove the R_{11} such that we get a solution graph which represents all values for ε_{11} for which an R_{11} exists such that

$$0 = -(\widetilde{R_{11}} \lll 30 - R_{11} \lll 30) + 1.$$

This solution graph shows that only $\varepsilon_{11} \in \{1, 4, 5\}$ is possible. Then by inserting each of these values in the original solution graph (by Algorithm 6) and counting the possible solutions for R_{11} (by Algorithm 4) it can be seen that $\varepsilon_{11} = 4$ is the best choice. Having fixed $\varepsilon_{11} = 4$ also the last but one equation includes only one of the ε_i, namely ε_{10} (implicitly in $\widetilde{R_{10}}$). Then possible solutions for ε_{10} can

be derived similarly as before for ε_{11} and doing this repeatedly gave us some good choices for $\varepsilon_{11}, \varepsilon_{10}, \varepsilon_9, \varepsilon_8, \varepsilon_7$ and (using the first two equations) for ε_3 and ε_4.
Finding values ε_5 and ε_6 such that the whole system still remains solvable was quite hard and could be done by repeatedly applying some of the techniques described in this paper, e.g. by combining generalized solution graphs for different of the equations and removing those variables R_i from the graphs which were no longer of any explicit use. This way we found four possible values for ε_5 and ε_6.

After fixing all the ε_i variables in a second step we were then able to construct the generalized solution graph for the complete system of equations with the remaining variables $R_1, \dots, R_{13}$. It contains about 700 vertices, more than 80000 edges and represents about 2^{205} solutions.

The second examplary system of equations appeared when looking for a so-called "connection" and after some reduction steps it can be written as follows:

$$\begin{aligned}
C_1 &= R_9 + \mathrm{Ch}(R_{12} \lll 2, R_{11}, R_{10}) \\
C_2 &= (C_3 - R_{10} - R_{11}) \oplus (C_4 + R_9 \lll 2) \\
C_5 &= (C_6 - R_{11}) \oplus (C_7 + R_{10} \lll 2 - (R_9 \lll 7)) \\
C_8 &= (C_9 - R_{12}) \oplus (C_{10} + R_9 \lll 2) \\
&\qquad\qquad \oplus (C_{11} + R_{11} \lll 2 - (R_{10} \lll 7) - \mathrm{Ch}(R_9 \lll 2, C_{12}, C_{13}))
\end{aligned}$$

In these equations the C_i are constants which come from some transformations of the original (quite large) system of equations together with some random choices of values. For this system we are interested in finding at least one solution for $R_9, R_{10}, R_{11}, R_{12}$.

As the first three equations are quite simple and (after eliminating the rotations) also quite narrow, the idea for solving this system was the following: First compute a generalized solution graph for the first three equations which represents all possible solutions for R_9, R_{10}, R_{11} for which at least one corresponding value for R_{12} exists. For this set of solutions we observed numbers of about 2^{11} to 2^{15} solutions. Then we could enumerate all these solutions from this graph and for each such solution we just had to compute the value for R_{12} corresponding to the last equation

$$\begin{aligned}
R_{12} = C_9 - (C_8 &\oplus (C_{10} + R_9 \lll 2) \\
&\oplus (C_{11} + R_{11} \lll 2 - (R_{10} \lll 7) - \mathrm{Ch}(R_9 \lll 2, C_{12}, C_{13})))
\end{aligned}$$

and check whether it also fulfilled the first equation. If we consider the first equation with random but fixed values for R_9, R_{10}, R_{11} we see that either there is no solution or there are many solutions for R_{12}, as only every second bit of R_{12} (on average) has an effect on the result of $\mathrm{Ch}(R_{12} \lll 2, R_{11}, R_{10})$. However, since the values for R_9, R_{10}, R_{11} were chosen from the solution graph of the first three equations there is at least one solution and thus the probabiliy that the value for R_{12} from the last equation also fulfills the first, is quite good.

This way we succeded in solving this system of equations quite efficiently.