Automated Search for Round 1 Differentials for SHA-1: Work in Progress

Philip Hawkes[1], Michael Paddon[1], and Gregory G. Rose[2]

[1] Qualcomm Australia, Level 3, 230 Victoria Rd, Gladesville, NSW 2111, Australia
{phawkes,mwp}@qualcomm.com
[2] QUALCOMM Incorporated, 5775 Morehouse Drive, San Diego, CA 92121 USA ggr@qualcomm.com

Abstract. Wang, Yin and Yu [24] describe a high probability differential path for SHA-1. Since then, various researchers have proposed techniques for improving the speed of finding collisions based on existing differential paths. The speed could be further improved by finding differentials through Round 1 (the first 20 steps) that are optimized to the particular technique. This paper describes progress on an automated search for finding differential paths through Round 1.
Keywords: Hash functions, Collision search attacks, SHA-1.

1 Introduction

Given the nature of this workshop, the cryptographic hash functions MD5 [16] and SHA-1 [5] need no introduction. A common element of the recent analyses of these hash function has been the combined description of the XOR differences and additive differences between the value of variable while processing two distinct messages: we call this description the *XOR-additive difference* for the variable. The differential paths consist of a sequence of XOR-additive differences, and a set of conditions required to ensure differences propagate correctly through the IF function.

The extensions[1] of the original attack [21, 22] on MD5 and the original attack [24] on SHA-1 use the original differential paths from [21, 22, 24].[2] Most research has focussed on improving how the original differential paths can be exploited, rather than looking for other differentials. Perhaps the underlying reason for this focus is the undeniable complexity of the existing differential path in Round 1 (steps 1-20). This differential must result in the input difference to Round 2 having a precise form. It is difficult enough to to verify why the conditions for the differential path in Round 1 are required, but it is much more difficult to find a completely new differential path that results in the correct input difference to Round 2. The aim of this paper is to share the author's experience of attempting to find new differential paths through the first round of SHA-1. This paper will ignore the differential path through rounds 2-4.

The intention of this research is to provide a search method that, when given the following inputs, will produce a differential path that satisfies the requirements specified in the input. The inputs are:

- The XOR differences in the expanded message words m_{i-1} for steps 1 to 20.
- The additive differences in $(a_0, b_0, c_0, d_0, e_0)$ at the start of Round 1.
- The XOR differences in $(a_{20}, b_{20}, c_{20}, d_{20}, e_{20})$ at the end of Round 1.

Our first attempt to find a differential path was a "by-hand" approach assisted by a spreadsheet. The user selects the differential path, and the spreadsheet ensures that, whenever any difference or value is assigned to a_i, then the difference or value is accurately passed to b_{i+1}, c_{i+2}, d_{i+3} and e_{i+4}. The attempts to find new 20-step differentials by hand failed, although the differentials often got within a one bit difference of what was required.

This propelled the development of an automated search for differential paths. As of this point in time, the implementation is incomplete, and few conclusions can be drawn. This paper outlines the approach used for this automated search, and is a report on work in progress.

Our approach is to create two search trees. The leaves of the *forward* search tree will correspond to *forward* differential paths between step 1 and step i that can be generated given specified differences in the input to step 1, and specified differences in the expanded message sequence. The leaves of the *reverse* search tree will correspond to *reverse* differential paths between step 20 and step i that can be generated given specified XOR and additive differences in the input to step 21, and specified differences in the expanded message sequence. We can then compare the differential paths in the leaves of the two trees to look for a match. This will make a differential path from step 1 through to step 20.

At this point in time, the forward search and reverse search have been implemented, while the

[1] See Section 2.1 for a quick summary.
[2] A technique for obtaining new differentials for SHA-1 is mentioned in [20], but no new differential paths are presented therein.

method for matching forward and reverse paths is yet to be implemented.

1.1 Outline of this paper

Section 2 contains a brief description of round 1 of SHA-1, summarizes previous research into MD5 and SHA-1 and describes the notation. Section 3 describes the basic branching components used in the forward and reverse search trees. The issue of generating the set of XOR-additive differences corresponding to a fixed additive difference is addressed in Section **??**. Section 5 describes the static branching, which is the most complicated part of the search. Section 8 describes how forward and reverse differential paths can be compared to find a match.

2 Description of SHA-1

Where possible, this description mirrors the notation in [23], and describes only those portions of SHA-1 required to understand the current paper. The SHA algorithms have four phases; padding, parsing, message scheduling and register update. The register-update function is called the *step function* in this paper. The padding and parsing takes a message of less than 2^{64} bits and creates a sequence of 512-bit *message blocks*. The padding and parsing are beyond the scope of this paper. The message scheduling expands the message block into a sequence of 80 *expanded message words* m_i, $0 \leq i \leq 79$. The step function is applied 80 times, processing the message blocks in sequence to transform a 160-bit input chaining variable into a 160-bit output chaining variable. The 80 steps are divided into four Rounds of 20 steps each. This paper addresses only Round 1 (the first 20 of these steps).

Message expansion: The message block is divided into sixteen 32-bit words which are assigned to $(m_0, \ldots, m_{15})$. The remaining message words m_i, $16 \leq i \leq 79$, are obtained iteratively as:

$$m_i = (m_{i-3} \oplus m_{i-8} \oplus m_{i-14} \oplus m_{i-16}) \lll 1.$$

Step Function for Round 1 (Steps 1-20): At step i, the step function uses the expanded message word m_{i-1} to transform five input state variables $(a_{i-1}, b_{i-1}, c_{i-1}, d_{i-1}, e_{i-1})$, into five outputs $(a_i, b_i, c_i, d_i, e_i)$. The initial state $(a_0, b_0, c_0, d_0, e_0)$ is obtained by partitioning the input chaining variable. For Steps 1 to 20, the step function generates the next state as follows:

$$\begin{aligned} a_i &= ROTL(a_{i-1}, 5) + f_{i-1} + e_{i-1} \\ &\quad + m_{i-1} + k_i \\ \text{where } f_{i-1} &= IF(b_{i-1}, c_{i-1}, d_{i-1}) \\ &= (b_{i-1} \wedge c_{i-1}) \vee ((\neg b_{i-1}) \wedge d_{i-1}); \\ b_i &= a_{i-1}; \\ c_i &= ROTL(b_{i-1}, 30); \\ d_i &= c_i; \\ e_i &= d_i. \end{aligned}$$

where $k_i = \texttt{0x5a827999}$. Steps 21 to 80 differ from Steps 1 to 20 in the function used to combine $b_{i-1}, c_{i-1}, d_{i-1}$, and the value of the constant k_i. The function f_{i-1} acts bitwise: that is $f_{i-1}[j]$ depends only on $b_{i-1}[j]$, $c_{i-1}[j]$ and $d_{i-1}[j]$.

2.1 Previous Work: A Quick Summary

Here is a quick summary of the previous analysis of MD5 and SHA-1.
MD5:

- Wang *et al* [21] presented collisions at the Crypto 2004 Rump Session; the details are found in the EUROCRYPT 2005 paper [22].
- Various researchers implemented the collision search [13, 9], some noting additional improvements.
- Some researchers have focussed on methods for improving the message modification method [10, 11, 17, 19].
- Others have sought to more accurately describe the necessary conditions [25, 18].

SHA-1.

- Biham and Chen [1] published results on SHA-0 at CRYPTO'2004.
- At the CRYPTO'2004 rump session, Joux [7] presented a SHA-0 collision, while Biham and Chen [2] presented independent results on full SHA-0 and reduced SHA-1 (SHA-1 with a reduced number of steps). The techniques (jointly published at EUROCRYPT 2005 [3]) are based on following a differential path of minimal weight across all steps of the hash function.
- At CRYPTO 2005, Wang, Yin and Yu presented a new approach for finding collisions of SHA-0 [23] and SHA-1 [24] (including 58-step reduced SHA-1). This approach uses a differential path of minimal weight for Rounds 2-4 and a high-weight differential path through the first round.

- Julta and Patthak [8] show that the differential path in [24] has minimal weight in Rounds 2-4.
- The paper of Sugita, Kawazoe and Imai [20] improves on the message modification technique by using using Guassian elimination. The approach of [20] using "neutral bits" and "semi-neutral bits" shares similarities with the "tunnels" approach applied by Klima [10] to MD5.

The various approaches to improving the collision search (in particular, the various message modification techniques) have the effectiveness limited by the conditions on the internal variables required for the differential path through Round 1. The message modification techniques could achieve a greater advantage if other differential paths (through Round 1) could be found. For example, there may be differential paths that place fewer conditions on the internal variables, or differential paths that are (in some way) more optimized for the particular technique This is one of the motivations for this research. Our research has not yet reached the point where differential paths can be optimized for particular message modification techniques.

2.2 Notation

The word size for SHA-1 is $n = 32$. A differential analysis follows the differences between internal values that occurs when processing two message blocks M' and M''. For an internal variable denoted (for example) by x, the value of the internal variable when processing M' is denoted by x', and the value of the internal variable when processing M'' is denoted by x''. When processing M' and M'', we write

$$\Delta_+ x = x'' - x' \pmod{2^n} \in \mathbb{Z}_{2^n};$$
$$\Delta_\oplus x = x'' \oplus x' \ \in GF(2^n);$$

$\Delta_+ x$ is the *additive difference* in x, while $\Delta_\oplus x$ is the *XOR difference* in x, We need a way to represent conditions on values x' and x'' of the form

- $\Delta_+ x \in Z_{2^n}$ is fixed to a known value;
- $\Delta_\oplus x \in GF(2^n)$ is fixed to a known value;
- the values of some bits where $x'[j] = x''[j] \in 0, 1$ are fixed to a known value.

To represent such conditions efficiently, we use a notation we have called *nabla* (∇) notation.[3] Nabla notation uses a symbol in the set $Q = \{$`@`, `+`, `-`, `*`, `0`, `1`$\}$ to represent information about each bit position of x' and x''. For a particular bit position j, the following notation is used:

- $\nabla x[j] \hookrightarrow$ `@`: assign $(x'[j], x''[j]) \in \{(0,1),\ (1,0)\}$
- $\nabla x[j] \hookrightarrow$ `+`: assign $(x'[j], x''[j]) = (0,1)$;
- $\nabla x[j] \hookrightarrow$ `-`: assign $(x'[j], x''[j]) = (1,0)$;
- $\nabla x[j] \hookrightarrow$ `*`: assign $x'[j] = x''[j]$;
- $\nabla x[j] \hookrightarrow$ `0`: assign $x'[j] = x''[j] = 0$;
- $\nabla x[j] \hookrightarrow$ `1`: assign $x'[j] = x''[j] = 1$.

Note that $\nabla x[j] \hookrightarrow$ `@` is equivalent to assigning the condition $\Delta_\oplus x[j] = 1$, while $\nabla x[j] \hookrightarrow$ `*` is equivalent to assigning the condition $\Delta_\oplus x[j] = 0$.

For a vector $\alpha \in Q^n$, and variable $x \in GF(2^n)$, $\nabla x \hookrightarrow \alpha$ denotes the n conditions $\nabla x[j] \hookrightarrow \alpha[j]$, $j \in [0, n-1]$ and α is called a *nabla representation* for the variable x.

For a vector $\alpha \in Q^n$, the corresponding XOR difference, denoted $\alpha_\oplus$ has $\alpha_\oplus[j] = 1$ if $\alpha[j] \in \{$`@`,`+`,`-`$\}$, and $\alpha_\oplus[j] = 0$ otherwise. That is, if $\nabla x \hookrightarrow = \alpha$, then $\Delta_\oplus x = \alpha_\oplus$. Thus a nabla representation α for x always fully specifies $\Delta_\oplus x$. The bit positions j where $\nabla x[j] \in \{$`@`,`+`,`-`$\}$ are the *dynamic bit positions* of x, and the bit $x[j]$ is said to be dynamic. The bit positions j where $\nabla x[j] \in \{$`*`,`0`,`1`$\}$ are the *static bit positions* of x, and the bit $x[j]$ is said to be static.

For α with $\alpha[j] \neq$ `@`, $\forall j \neq (n-1)$, the corresponding additive difference, denoted α_+ can be computed as follows: if $\alpha[n-1] \in \{$`*`,`0`,`1`$\}$ then

$$\alpha_+ = \sum_{j:\alpha[j] \in \{\texttt{+},\texttt{-}\}} \alpha[j] \cdot 2^j \pmod{2^n},$$

while if $\alpha[n-1] \in \{$`@`,`+`,`-`$\}$, then

$$\alpha_+ = 2^{n-1} + \sum_{j:\alpha[j] \in \{\texttt{+},\texttt{-}\}} \alpha[j] \cdot 2^j \pmod{2^n},$$

Thus, provided a nabla representation α for x has $\alpha[j] \neq$ `@`, $\forall j \neq (n-1)$, then the nabla representation fully specifies $\Delta_+ x$. In most cases in this analysis, the nabla representation will always fully specify $\Delta_+ x$. Note also that if $y = ROTL(x, r)$, then $\nabla y \hookrightarrow ROTL(\nabla x, r)$.

The nabla notation is useful for representing the propagation of additive and XOR differences through the SHA-1 step function because of the properties preserved by addition and rotation, and the ability to analyse the nabla representation of the inputs and outputs of the IF function on a bit-by-bit basis.

3 Search Trees

Recall that our goal is to create a forward search tree and a reverse search tree, and then compare differential paths in the leaves of the two trees to look for a match.

[3] The term "nabla" has been used because `\nabla` is the LaTex2e command for the symbol ∇.

The branches and leaves of a sub-tree starting from a particular branching point will satisfy the set of conditions that have been applied up to that branching point. Each branching point corresponds to adding a new condition to the set of existing conditions. As the search tree searches through the sub-branches, the search must ensure that the existing conditions are not contradicted. For example, once an additive difference in a particular variable is assigned for a particular branch, then all branches in the sub-tree proceeding from that branch must respect that condition. The search tree must respect those conditions applied below the branching point.

In terms of the nabla notation, this means that once a condition of the form $\nabla x[j] \in \{+,-,0,1\}$ is assigned, then this condition cannot be altered in the sub-tree above where the assignment occurs. However, if a condition of the form $\nabla x[j] \hookrightarrow$ "`@`" is assigned, then this condition can be refined to $\nabla x[j] \hookrightarrow$ "`+`" or $\nabla x[j] \hookrightarrow$ "`-`" at a branching point in the sub-tree above where the assignment occurs. Similarly , if a condition of the form $\nabla x[j] \hookrightarrow$ "`*`" is assigned, then this condition can be refined to $\nabla x[j] \hookrightarrow$ "`0`" or $\nabla x[j] \hookrightarrow$ "`1`" at a branching point in the sub-tree above where the assignment occurs.

There are also variables for which the conditions only specify the additive difference $\Delta_+ x$, rather than assigning conditions to individual bits $\nabla x[j]$. This is discussed in further detail below.

3.1 Types of Branches in the Search Tree

There are four types of branching points in these search trees:

XOR-to-Additive Branching: This occurs when the XOR difference in a particular variable has been specified for this sub-tree, but there are variety of additive differences that are still possible. For example, at the beginning of the search we specify the bits of the message words for which there is an XOR difference: if there is an XOR difference at position j, then we specify $\mu_{i-1}[j] \hookrightarrow$ "`@`"; if these is no XOR difference at position j, then we specify $\mu_{i-1}[j] \hookrightarrow$ "`*`". For those bit positions j where we specify $\mu_{i-1}[j] \hookrightarrow$ "`*`", a further refinement to $\mu_{i-1}[j] \hookrightarrow$ "`0`", or $\mu_{i-1}[j] \hookrightarrow$ "`1`", has no effect on the additive difference in m_{i-1}, so there is no point in creating separate paths for $\mu_{i-1}[j] \hookrightarrow$ "`0`", and $\mu_{i-1}[j] \hookrightarrow$ "`1`". However, for those bit positions where $\mu_{i-1}[j] =$ "`@`", then a refinement $\mu_{i-1}[j] \hookrightarrow$ "`+`", or $\mu_{i-1}[j] \hookrightarrow$ "`-`", has a significant effect on the additive difference in m_{i-1} (MSB $j = (n-1)$ excepted.) Thus, for each bit position $j < (n-1)$ where $\mu_{i-1}[j] =$ "`@`", we get a branching into two sub-trees. If there are `numdiff` such bit positions, then we get a branching into $2^{\texttt{numdiff}}$ sub-trees. Note that XOR-to-Additive branching places no conditions on static bits, other than specifying which bit positions are static.

Additive-to-XOR Branching: This occurs when the additive difference in a particular variable has been specified for this sub-tree, but there are variety of XOR differences (and corresponding nabla representations) that are still possible. To determine the additive difference in the output f_{i-1} from the IF function requires (in most cases) the nabla representation of all input variables be specified. Hence, the search tree should search all possible XOR differences available for the input differences. For example, if the search tree has assigned $\Delta_+ a_{i-1} = +1$, then branching points in the corresponding sub-tree can correspond to nabla representations for ∇b_i are of the form:

- (`*******.*+`): a single "`+`" in the LSB with all "`*`" to the left);
- (`*..*+--..-`): a single "`+`" with all "`*`" to the left and all "`-`" to the right;
- (`+------..-`): a single "`+`" in the MSB with all "`-`" to the right; and
- (`-------..-`): all "`-`", which has the same XOR difference as the previous nabla representation

Section 4 describes the method used to find the nabla representations corresponding to a given additive difference. Note that an Additive-to-XOR branching places no conditions on static bits, other than specifying which bit positions are static.

Static Branching: This branch occurs as a result of the IF function. If the value(s) of static bits of b_{i-1}, c_{i-1}, d_{i-1} can affect the additive difference int the output f_{i-1} from the IF function, and if those static bits have not already been assigned, then sub-branches can be created for the possible choice of the static bits. This branching is very complex; the details discussed in a separate section (Section 5). Note that the values of static bits affect the differential path only through the IF function.

Additive-Rotation Branching: This branch occurs only with the forward tree search. This occurs when the additive difference in a_{i-1} has been specified for this sub-tree, but there are a variety of possible additive differences for $ROTL(a_{i-1} i, 5)$. For example, the following

nabla representations for $ROTL(a_{i-1}, 5)$ correspond to $\Delta_+ a_{i-1} = +1$:

- $\nabla a_{i-1} \hookrightarrow$ (**..*+) will result in $\nabla ROTL(a_{i-1}, 5) \hookrightarrow$ (**..*++++++), with additive difference $\Delta_+ ROTL(a_{i-1}, 5) = 2^5$. Similarly,(**..*+-), (**..*+--),...,(*****+--..-), also result in $\Delta_+ ROTL(a_i, 5) = 2^5$.
- $\nabla a_i \hookrightarrow$ (-..--) will result in nabla representation $\nabla ROTL(a_i, 5) \hookrightarrow$ (-..--) which has $\Delta_+ ROTL(a_{i-1}, 5) = 1$.
- The remaining nabla representations, such as $\nabla a_{i-1} \hookrightarrow$ (****+--..-), result in $\Delta_+ ROTL(a_{i-1}, 5) = 1 + 2^5$.

There are only a few possible values for $\Delta_+ ROTL(a_{i-1}, 5)$ corresponding to a particular additive difference $\Delta_+ a_{i-1}$. The additive difference in $\Delta_+ ROTL(a_{i-1}, 5)$ will affect additive differences in $\Delta_+ a_{i+1}$; thus, the choice of ∇a_i can result in a variety of additive differences in $\Delta_+ a_i$. Each of the possible additive differences $\Delta_+ a_i$ needs to be explored, so our search tree includes a branch for each possible value of $\Delta_+ ROTL(a_{i-1}, 5)$.

The Additive-to-XOR, XOR-to-Additive and Additive-Rotation branching are *dynamic branchings* because these branchings only consider the allocation of dynamic bits.

3.2 The Forward Search Tree

The forward search tree determines the possible additive differences $\Delta_+ a_i$ coming out of step i, and for each sub-branch, the search tree proceeds to the following step (step $(i+1)$) to determine the possible additive differences $\Delta_+ a_{i+1}$. The forward search tree uses the relation:

$$\Delta_+ a_i = \Delta_+ ROTL(a_{i-1}, 5) + \Delta_+ m_{i-1} + \Delta_+ e_{i-1} + \Delta_+ f_{i-1}. \quad (1)$$

The following algorithm can be used to "add" two nabla representations to get a third nabla representation that has additive difference equal to the sum of the input additive differences. This can be preferable to converting the nabla representations to an additive difference, and makes it easier to see where differences are occuring.

Procedure: Add α, β to produce $\omega = \alpha \boxplus \beta$
Inputs: Nabla representations `alpha[],beta[]`
Outputs: Nabla representation `omega[]`
Algorithm

```
int carry=0;
int j;
int a_rep, b_rep, carry;

for( j=0; j<n; j++ ){
  switch( alpha[j] ){
    case '@':   a_rep= 1; break;  /*MSB*/
    case '+':   a_rep= 1; break;
    case '-':   a_rep=-1; break;
    default:    a_rep= 0; break;
  }
  switch( beta[j] ){
    case '@':   b_rep= 1; break;  /*MSB*/
    case '+':   b_rep= 1; break;
    case '-':   b_rep=-1; break;
    default:    b_rep= 0; break;
  }
  switch( a_rep + b_rep + carry ){
    case -3: omega[j]='-'; carry=-1; break;
    case -2: omega[j]='*'; carry=-1; break;
    case -1: omega[j]='-'; carry= 0; break;
    case -0: omega[j]='*'; carry= 0; break;
    case  1: omega[j]='+'; carry= 0; break;
    case  2: omega[j]='*'; carry= 1; break;
    case  3: omega[j]='+'; carry= 1; break;
  }
}
return( omega )
```

3.3 The Reverse Search Tree

Like the forward search tree, the reverse search tree consists of branches alternating between reverse dynamic branching components and static branching components. The reverse search treedetermines the possible additive differences $\Delta_+ e_{i-1}$ going into Step i, and for each sub-branch, the search tree proceeds to the following step (Step $(i-1)$) to determine the possible additive differences $\Delta_+ e_{i-2}$. The reverse search tree uses the relation:

$$\Delta_+ e_{i-1} = \Delta_+ a_i - \Delta_+ ROTL(a_{i-1}, 5) - \Delta_+ m_{i-1} - \Delta_+ f_{i-1}.$$

The modular subtraction can be implemented using the nabla representation addition "$\boxplus$" (described above in Section 6) in combination with a negation operation (negation is achieved by reversing the sign of all dynamic bits).

Each step examined by the forward tree and reverse tree includes a dynamic branching component (including Additive-Rotation branching, XOR-to-Additive branching, and Additive-to-XOR branching) followed by a static branching component. The

conditions on b_{i-1}, c_{i-1} and d_{i-1} prior to static branching are denoted using $\nabla b_{i-1}^{\text{in}}$, $\nabla c_{i-1}^{\text{in}}$ and $\nabla d_{i-1}^{\text{in}}$, and the conditions after static branching are denoted $\nabla b_{i-1}^{\text{out}}$, $\nabla c_{i-1}^{\text{out}}$ and $\nabla d_{i-1}^{\text{out}}$.

3.4 Reconstruction Information

As the search algorithms generate forward and reverse paths, there are several reasons for recording information about the path. Firstly, in order to match forward paths to reverse paths we need to compare the differential path at the step where the two paths meet. Section 8 considers what "comparison information" needs to be stored to match forward and reverse paths.

Secondly, once a match is found between a forward and reverse path, we then need to reconstruct the full 20-step path. This "reconstruction information" does not need to be stored in the same location as the comparison information, provided the comparison information provides a means for locating the reconstruction information. The reconstruction information serves an addition purpose: if it is necessary to stop the forward search or reverse search in the middle of a search, then the reconstruction information provides a "last known state" from which the search can continue.

A differential path can be reconstructed if we can un-ambiguously record the choice of branches at each branch point. In the remainder of this section, we describe the method for recording branches for XOR-to-Additive branching, Additive-to-XOR branching and Additive-Rotation branching. Section 5 describes the method used to record branches for Static Branching.

XOR-to-Additive branching. Assume $\Delta_{\oplus} x$ has already been specified and $\Delta_{\oplus} x$ has weight w. To identify the branch corresponding to a value of $\Delta_{+} x$, it is sufficient to store the w signs ("+" or "−") for $\nabla x[j]$ at those bits where $\Delta_{\oplus} x[j] = 1$. This information can be stored in a w-bit integer. In some cases, the sign of an MSB difference does not need to be specified, while in other cases the MSB difference does need to be specified.

Additive-to-XOR Branching. The reconstruction information needs to identify which value of ∇x corresponding to a given $\Delta_{+} x$ is associated with the path. Our reconstruction method requires a well-defined ordering for the set of nabla representations (that is, a way to define when one nabla representation is greater than another). For a given additive difference, we generate and order the the corresponding set of nabla representation using our well-defined ordering. The reconstruction information used to identify an Additive-to-XOR branch is the position where the nabla representation occurs in the ordered list.

Note that in the forward search, ∇b_i must not only correspond to both the correct additive difference for a_{i-1}, but also correspond to the correct additive difference for $ROTL(a_{i-1}, 5)$.

Additive-Rotation Branching. Similar to the method used for the Additive-to-XOR branching we generate and order the possible values of $\Delta_{+} r_{i-1}$ corresponding to the additive difference for a_{i-1}. To indicate one of these additive-differences $\Delta_{+} r_{i-1}$, we store an index where the additive difference occurs in the list.

4 Additive-to-XOR Branching

This section considers the question: how do we create the set of nabla representations corresponding to a particular additive difference?

We can begin by writing the additive difference D modulo 2^n in binary representation $D = \sum D_j 2^j$, which then translates into a nabla representation $\nabla x[j] =$ + when $D_j = 1$ and $\nabla x[j] =$ * when $D_j = 0$. This is then "reduced" to a nabla representation with a smaller number of dynamic bits, but with equal additive difference using the following rules:

- A "*" followed by a sequence of consecutive "+" symbols can be translated into two "−" symbols: "*++" can become "−*+"; "*+++" can become "−**+"; "*++++" can become "−***+"; and so forth.
- "−+" can be translated to "*−".
- "+−" can be translated to "*+".
- If the MSB is dynamic, then the sign can be changed to see if this allows a reduction in the number of dynamic bits.

By testing combinations of the above translations, we obtain a "root" $\nabla x \hookrightarrow \overline{\chi}$ that has a smaller number of dynamic bits.

The next step is to use this root to find the possible nabla representations. If If the root $\overline{\chi}$ is w dynamic bits then we create an array $y[0], y[1], \ldots, y[w-1]$ that stores the indices of the dynamic bits in $\overline{\chi}$, with the indices arranged in increasing order. We can consider $\overline{\chi}$ as the sum of w component nabla representations, where the k-th component nabla representation is

$$X_k = (*, \ldots, *, \overline{\chi}[y[k]], *, \ldots, *)$$

If $\overline{\chi}[y[k]] =$ −, the we ca we can generate new nabla representations equivalent to X_k by recursively applying the translation "*−" → "−+". We let $X_k^{(j)}$ denote the equivalent component nabla representation

consisting of one "-" symbol followed by j "+" symbols. Similarly, if $\overline{\chi}[y[k]] = +$, then we let $X_k^{(j)}$ denote the equivalent component nabla representation consisting of one "+" symbol followed by j "-" symbols.

Basically, we have carried the addition difference to higher order bits. For each of $X_0, \ldots, X_{w-1}$, we can create alternative nabla representations (with identical additive difference) by adding together $(X_0^{(j[0])}, \ldots, X_{w-1}^{(j[w-1])})$, using the nabla representation addition algorithm in Section 6.

By generating all possible equivalent component nabla representations for each X_k, we will generate all possible alternative nabla representations corresponding to the addition difference $\Delta_+ x = \chi_+$. Once $j \geq n - y[i]$, the difference will have been carried passed the most significant bit, so it is clear that j must be upper bounded by $(n - y[i])$. However, the set of $\prod_{i=0}^{w-1}(n - y[i])$ combinations $(X_0^{(j[0])}, \ldots, X_{w-1}^{(j[w-1])})$ can result in duplicate alternative nabla representations (that is, two alternative nabla representations that are equal).

For a first example, consider a case with $X_0 =$ (..*..****-) and $X_1 =$ (..*..**-**). The alternative nabla representations produced by the above technique includes the following duplicate alternative nabla representations:

$$\begin{aligned}
(*..***-++) \boxplus (*..*****-**) &= (*..**-+++) \\
(*..**-+++) \boxplus (*..*****-**) &= (*..**-+++) \\
(*..***-++) \boxplus (*..****-+**) &= (*..**-+++) \\
(*..**-+++) \boxplus (*..****-+**) &= (*..**-+++)
\end{aligned}$$

Note that for $X_0^{(j[0])} \boxplus X_1^{(j[1])}$ we obtain no new representations by considering $j[0] > 2 = y[1] - y[0]$.

For a second example, consider a case with $X_0 =$ (..*..****-) and $X_1 =$ (..*..**+**). The alternative nabla representations produced by the above technique includes the following duplicate alternative nabla representations:

$$\begin{aligned}
(*..**-++) \boxplus (*..*******) &= (*..*****+) \\
(*..***-+) \boxplus (*..*******) &= (*..*****+) \\
(*..**-++) \boxplus (*..****+-**) &= (*..*****+) \\
(*..*-+++) \boxplus (*..****+-**) &= (*..*****+)
\end{aligned}$$

Once again, we obtain no new representations by considering $j[0] > 2 = y[1] - y[0]$. In the general case, carrying a difference past the next dynamic bit in the root representation (that is, using $j[i] > y[i+1] - y[i]$) can be shown to result in a duplicate nabla representation.

Additive-to-XOR Branching Algorithm

1. Find one nabla representation corresponding to the additive difference (for example, using the above technique at the beginning of this section: translating the binary representation of the additive difference to a nabla representation consisting only of + and * symbols).
2. Reduce this nabla representation to a root representation $\overline{\chi}$ with lower number of dynamic bits.
3. Create the array $(y[0], y[1], \ldots, y[w-1])$ containing the indices of the dynamic bits of the root representation $\overline{\chi}$.
4. For each combination of $(j[0], j[1], \ldots, j[w-1])$ in the range

$$\begin{aligned}
&0 \leq j[i] \leq y[i+1] - y[i],\ 0 \leq i \leq w-2; \\
&0 \leq j[w-1] \leq n - y[w-1].
\end{aligned}$$

compute and store the nabla representation $\boxplus_{i=0}^{w-1} X_i^{j[i]}$. □

5 Static Branching

At Step i, the static branching produces a set of branches corresponding to distinct values of ∇f_{i-1}. (For the remainder of Section 5, the sub-script $(i-1)$ for b, c, d, f shall be implicit and will not be written). The set of possible branches depends only on the input nabla representations ∇b^{in}, ∇c^{in}, ∇d^{in}.

As a by-product of the bit-wise nature of the IF function, the set of possible ∇f is produced by the combinations of possible values $\nabla f[j]$ for each bit position j, $0 \leq j \leq n-1$. That is, the static branching can be thought of as a set of n independent branch-points corresponding to the n bit positions j.

The choice of possible values $\nabla f[j]$ at bit position j depends only on the combination of $\nabla b^{\text{in}}[j]$, $\nabla c^{\text{in}}[j]$, $\nabla d^{\text{in}}[j]$. Each combination $\nabla b^{\text{in}}[j]$, $\nabla c^{\text{in}}[j]$, $\nabla d^{\text{in}}[j]$ is dealt with in one of the cases shown in Table 1 and Table 1 of Section 5.2. These tables show the branches that are possible for each case.

The complexity of the search algorithm is directly related to the number of branches at each branch point. It is essential that we minimize the number of branches at each branch point. We now apply this principle to the static branching. Recall that the purpose of determining ∇f is to determine the value of $\Delta_+ f$. The additive difference $\Delta_+ f$ is independent of static bit values of f: that is, the specific value of that static bit values is not relevant to the search for differentials. Therefore, there is no advantage to differentiating a branch with $\nabla f[j] =$ '*' from a branch with $\nabla f[j] =$ '0' or a branch with $\nabla f[j] =$ '1', since all three branches result in the same value of $\Delta_+ f$.

In order to reduce the number of branches searched, all possible static values of $\nabla f[j]$ (for a given case) are grouped into a single branch. Similarly, when examining the MSB ($j = n-1$), note that all dynamic values of $\nabla f[n-1]$produce identical values of $\Delta_+ f$. Hence, all possible dynamic values of $\nabla f[n-1]$ (for a given case) are grouped into a single branch.

For each j, the distinct branches are obtained by apply additional conditions to the input nabla representations $\nabla b^{\text{in}}[j]$, $\nabla c^{\text{in}}[j]$, $\nabla d^{\text{in}}[j]$: the resulting conditions are represented by ∇b^{out}, ∇c^{out}, ∇d^{out}.

5.1 Relational Conditions

In some cases it is only necessary to assign values to one static bit. For example, if $(\nabla b[j], \nabla c[j], \nabla d[j]) =$ (`*,+,-`), them $\nabla b[j] =$ "`0`", would result in $\nabla f[j] =$ "`-`", while $\nabla b[j] =$ "`1`", would result in $\nabla f[j] =$ "`+`").

In other cases it is necessary to assign values to two static bits. For example, if $\nabla b[j] \in$ {`+,-`}, and $(\nabla c[j], \nabla d[j]) =$ (`*,*`), then the four possible assignments of `0,1` to $\nabla c[j]$ and $\nabla d[j]$ will have the following effect of $\nabla f[j]$.

- $(\nabla c[j], \nabla d[j]) = (0, 0), \quad \Rightarrow \quad \nabla f[j] = 0.$
- $(\nabla c[j], \nabla d[j]) = (0, 1), \quad \Rightarrow \quad \nabla f[j] = -\nabla b[j].$
- $(\nabla c[j], \nabla d[j]) = (1, 0), \quad \Rightarrow \quad \nabla f[j] = \nabla b[j].$
- $(\nabla c[j], \nabla d[j]) = (1, 1), \quad \Rightarrow \quad \nabla f[j] = 1.$

The values of $\nabla f[j]$ when $\nabla f[j] \notin$ {`+,-`} have no effect on $\Delta_+ f$ (which, after all, is the important value). Thus, the two cases $(\nabla c[j], \nabla d[j]) = (0, 0)$ and $(\nabla c[j], \nabla d[j]) = (1, 1)$ have the same effect on $\Delta_+ f$. Rather than consider the cases $(\nabla c[j], \nabla d[j]) = (0, 0)$ and $(\nabla c[j], \nabla d[j]) = (1, 1)$ as distinct branches of the search tree, it is more efficient to assign $\nabla c[j] = \nabla d[j]$ for this sub-tree, and remember this while we search in that sub-tree. There are other situations where is desirable to assign $\nabla c[j] \neq \nabla d[j]$ for a branch, and remember this while we search in that sub-tree. We term these "relational conditions".

Relational conditions introduce additional complexity because it is not obvious how the search should remember which relational conditions have already been assigned. To implement this, we use two arrays `v0[]` and `v1[]` and we keep a counter `vc` that provides the index of the next unused elements in the arrays `v0[]` and `v1[]`. The principle behind these two arrays is that if a value is ever assigned to `v0[index]`, then `v1[index]` is automatically assigned the value `v1[index]` $= 1 -$ `v0[index]`. We initialize `v0[]` with lower case letters in order, and initialize `v1[]` with upper case letters in order, with the knowledge that the value of a lower case letter is the opposite value of the corresponding lower case letter. For each state variable x we assign an array to store ∇x and an integer array $xp[n]$.

- If bit j of variables x is completely specified (that is, $\nabla x[j] \in$ {`+,-,0,1`}), then this value is stored in the array for ∇x and we assign $xp[j] = 0$.
- If bit j of variables x is static, but undefined and not related to any other variable, then we assign $\nabla x[j] =$ "`*`" and assign $xp[j] = 0$.
- If we wish to assign a condition where two static values, for example $\nabla c[j]$ and $\nabla d[j]$ are equal (but not specified to be 0 or 1), then we assign $cp[j] = dp[j] =$ `+vc`. The existence of a non-zero integer in $cp[j]$ is an indicator that the value is related to another bit value. A positive value $cp[j] = k$ indicates that the search should look in `v0[k]` to find information about $cp[k]$, while a negative value in $cp[j] = k$ indicates that the search should look in `v1[k]` to find information about $cp[k]$.
  - If the search later need to assign a condition equivalent to specifying $\nabla c[j] = 1$ or $\nabla d[j] = 1$, then the search simply assigns `v0[k]=1`, and `v0[k]=0`. Thereafter, the search will read that both $c[j]$ and $d[j]$ have $\nabla c[j] = \nabla d[j] = 1$. Likewise if the search needs to assign $\nabla c[j] = 0$ or $\nabla d[j] = 0$.
  - It the search later needs to assign some bit position j' of another variable x as also equal to $\nabla c[j]$ or $\nabla d[j]$, then the search can copy the value from $cp[j]$ to $xp[j']$.
  - It the search later needs to assign some bit j' of another variable x as different to $\nabla c[j]$ or $\nabla d[j]$, then the search can set $xp[j'] = -cp[j]$. This will then indicate to the search that the value of $xp[j']$ is stored in `v1[k]`.
- If the search wishes to assign a condition where two static values, for example $\nabla c[j]$ and $\nabla d[j]$ are not equal (but not specified to be specific values), then the search assigns $cp[j] =$ `+vc` and $dp[j] =$ `-vc`, indicating that information about $c[j]$ is found in `v0[vc]`, while information about $d[j]$ is found in `v1[vc]`. If the search needs to apply specify values to $d[j]$ or $c[j]$ at a later branching, then the same approach can be used as described above.

5.2 Details of the Static Branching

Now that we have a method for remembering the relational conditions, we can now describe the various cases for Static Branching.

Table 1 and Table 2 contain all the necessary details for possible combinations of values $\nabla(b[j], c[j], d[j])$. Table 1 lists the cases where $b[j]$ is

dynamic. Note that the sign of a dynamic bit of an input to the IF function is always defined, so Table 1 presumes that $b[j] \in \{$`+,-`$\}$ (that is $b[j] \neq$ '`@`'). Table 2 lists the cases where $b[j]$ is static. The considerations for the static branching are quite complicated, but quite easy to implement once the rules are explicitly written out.

Each case allows between 1 and 3 branches; we denote the number of branches for position j by $Br[j]$. For each position j, each branch has a corresponding branch index $BI[j]$ in the range $[0, Br[j]]$,for each branch occurring at each bit position. Table 1 and Table 2 lists the number of branches $Br[j]$ for each case, and the branch index assigned to each branch. A record of the n static branch indices is sufficient to determine $\Delta_+ f$ and assign any additional conditions required for ∇b^{out}, ∇c^{out} and ∇d^{out}.

We briefly discuss some cases to explain the tables.

Case 1: $b[j]$ is dynamic. In this case, $\nabla f[j]$ is affected by the values of both $\nabla c[j]$ and $\nabla d[j]$.

- **Case 1a:** $\nabla c^{\text{in}}[j] \in \{$`+,-,0,1`$\}$ and $\nabla d^{\text{in}}[j] \in \{$`+,-,0,1`$\}$. Since all input are fully specified, the output $\nabla f[j]$ is also fully specified and there is only one branch.
- **Case 1b:** $\nabla c^{\text{in}}[j] =$ '`*`' and $\nabla d^{\text{in}}[j] \in \{$`+,-,0,1`$\}$. In this case the value of $\nabla f[j]$ is affected by the choice of value for $\nabla c^{\text{out}}[j] \in \{$`0,1`$\}$. There is one branch with $\nabla c^{\text{out}}[j] =$ '`0`' and another branch with $\nabla c^{\text{out}}[j] =$ '`1`'. Case 1c is similar.
- **Case 1d (Not MSB):** $\nabla c^{\text{in}}[j] = \nabla d^{\text{in}}[j] =$ "`*`", $0 \leq j \leq n-2$. In this case there are three branches:
  - $(\nabla c^{\text{out}}[j], \nabla d^{\text{out}}[j]) =$ (`v0[vc]`, `v0[vc]`); $\Rightarrow$ $\nabla f[j] =$ '`*`'.
  - $(\nabla c^{\text{out}}[j], \nabla d^{\text{out}}[j]) =$ (`1`, `0`); $\Rightarrow$ $\nabla f[j] = \nabla b[j]$.
  - $(\nabla c^{\text{out}}[j], \nabla d^{\text{out}}[j]) =$ (`0`, `1`); $\Rightarrow$ $\nabla f[j] = -\nabla b[j]$.
- **Case 1d (MSB):** $\nabla c^{\text{in}}[j] = \nabla d^{\text{in}}[j] =$ '`*`'. In this case there are only two branches:
  - $(\nabla c^{\text{out}}[j], \nabla d^{\text{out}}[j]) =$ (`v0[vc]`, `v0[vc]`); $\Rightarrow$ $\nabla f[j] =$ '`*`'.
  - $(\nabla c^{\text{out}}[j], \nabla d^{\text{out}}[j]) =$ (`v0[vc]`, `v1[vc]`) This makes $f[j]$ dynamic without specifying the sign, which is acceptable for the MSB.
- **Case 1e:** $\nabla c^{\text{in}}[j] \in \{$`+,-,0,1`$\}$ and $\nabla d^{\text{in}}[j] =$ `v0[x]`. This case occurs only in the forward differential. The two branches correspond to assigning `v0[x]=0` and `v0[x]=1`. Case 1e, 1i and 1j are treated similarly.

Case	Input ∇				New Conditions				
	$c[j]$	$d[j]$	Br	BI	$c[j]$	$d[j]$	`v0`	`v1`	$\nabla f[j]$
1a	`+,-,0,1`	`+,-,0,1`	1	0	.	.	.	.	Det.
1b	`*`	`+,-,0,1`	2	0	0	.	.	.	Det.
				1	1	.	.	.	Det.
1c	`+,-,0,1`	`*`	2	0		. 0	.	.	Det.
				1	.	1	.	.	Det.
1d: Not MSB	`*`	`*`	3	0	`v0`	`v0`	.	.	`*`
				1	1	0	.	.	$\nabla b[j]$
				2	0	1	.	.	$-\nabla b[j]$
1d: MSB	`*`	`*`	2	0	`v0`	`v0`	.	.	`*`
				1	`v0`	`v1`	.	.	`@`
1e	`+,-,,0,1`	`v0`	2	0	.	.	0	(1)	Det
				1	.	.	1	(0)	Det
1f	`+,-,,0,1`	`v1`	2	0	.	.	(1)	0	Det
				1	.	.	(0)	1	Det
1g: Not MSB	`*`	`v0`	3	0	`v0`	.	.	.	`*`
				1	.	1	0	(1)	$\nabla b[j]$
				2	.	0	1	(0)	$-\nabla b[j]$
1g: MSB	`*`	`v0`	2	0	`v0`	.	.	.	`*`
				1	`v1`	.	.	.	`@`
1h: Not MSB	`*`	`v1`	3	0	`v1`	.	.	.	`*`
				1	.	1	(1)	0	$\nabla b[j]$
				2	.	0	(0)	1	$-\nabla b[j]$
1h: MSB	`*`	`v1`	2	0	`v1`	.	.	.	`*`
				1	`v0`	.	.	.	`@`
1i	`v0`	`+,-,0,1`	2	0	.	.	0	(1)	Det
				1	.	.	1	(0)	Det
1j	`v1`	`+,-,0,1`	2	0	.	.	(1)	0	Det
				1	.	.	(0)	1	Det
1k: Not MSB	`v0`	`*`	3	0	.	`v0`	.	.	`*`
				1	.	0	1	(0)	$\nabla b[j]$
				2	.	1	0	(1)	$-\nabla b[j]$
1k: MSB	`v0`	`*`	2	0	.	`v0`	.	.	`*`
				1	.	`v1`	.	.	`@`
1l: Not MSB	`v1`	`*`	3	0	.	`v1`	.	.	`*`
				1	.	0	(0)	1	$\nabla b[j]$
				2	.	1	(1)	0	$-\nabla b[j]$
1l: MSB	`v1`	`*`	2	0	.	`v1`	.	.	`*`
				1	.	`v0`	.	.	`@`

Table 1. Static Branching when $\nabla b^{\text{in}}_{i-1}[j]$ is dynamic (that is, $\nabla b^{\text{in}}_{i-1}[j] \in \{$`+,-`$\}$. In this table, "Det." denotes that $\nabla f_{i-1}[j]$ can be determined once the new conditions have been assigned. References to `v0` (`v1`) indicate a specific array entry `v0[x]` (`v1[x]`) respectively. In all cases in this table, $\nabla b^{\text{out}}_{i-1}[j] = \nabla b^{\text{in}}_{i-1}[j]$.

Case	Input ∇		Br	BI	$\nabla b_{i-1}^{\text{out}}[j]$	$\nabla f[j]$
	c	d				
$\nabla b_{i-1}^{\text{in}}[j] =$ '*'						
2a	Static	Static	1	0	*	*
2b	Static	Dyn.	2	0	0	$\nabla d[j]$
				1	1	Static
2c	Dyn.	Static	2	0	0	Static
				1	1	$\nabla c[j]$
2d (Not MSB)	$\nabla c[j] = \nabla d[j]$ Both Dyn.		1	0	*	$\nabla c[j]$
2e	$\nabla c[j] \neq \nabla d[j]$		2	0	0	$\nabla d^{\text{in}}[j]$
(Not MSB)	Both Dyn.			1	1	$\nabla c^{\text{in}}[j]$
2f (MSB)	Dyn.	Dyn.	1	0	*	Dyn.
$\nabla b_{i-1}^{\text{in}}[j] =$ '0'						
3a	Any	Any	1	0	.	$\nabla d[j]$
$\nabla b_{i-1}^{\text{in}}[j] =$ '1'						
3b	Any	Any	1	0	.	$\nabla c[j]$

Table 2. Static Branching when $\nabla b_{i-1}^{\text{in}}[j]$ is static. In this table, "Dyn." represents the condition that the bit position is dynamic. In all cases in this table, $\nabla c_{i-1}^{\text{out}}[j] = \nabla c_{i-1}^{\text{in}}[j]$ and $\nabla d_{i-1}^{\text{out}}[j] = \nabla d_{i-1}^{\text{in}}[j]$.

- **Case 1g (Not MSB):** $\nabla c^{\text{in}}[j]$ = '*' and $\nabla d^{\text{in}}[j] =$ `v0[x]`, $0 \leq j \leq n-2$. There are three branches in this case:
  - $\nabla c[j] =$ `v0[x]` $= \nabla d[j]$; $\Rightarrow \nabla f[j] =$ '*'.
  - $(\nabla c[j], v0[x]) = (1, 0)$; $\Rightarrow \nabla fi[j] = \nabla b[j]$.
  - $(\nabla c[j], v0[x]) = (0, 1)$; resulting in $\nabla fi[j] = -\nabla b[j]$.

  Cases 1h (Not MSB), 1k (Not MSB) and 1l (Not MSB) are treated similarly.
- **Case 1g (MSB):** $\nabla c^{\text{in}}[n-1] =$ '*' and $\nabla d^{\text{in}}[n-1] =$ `v0[x]`. There are two branches in this case:
  - $\nabla c[j] =$ `v0[x]` $= \nabla d[j]$; $\Rightarrow \nabla f[j] =$ '*'.
  - $\nabla c[j] =$ `v1[vc]` $= 1 - \nabla d[j]$; $\Rightarrow \nabla f[j] =$ '@'. Acceptable for the MSB

  Cases 1h (MSB), 1k (MSB) and 1l (MSB) are treated similarly.

Case 2: $\nabla b^{\text{in}}[j] =$ '*'. In this case, the search has assigned $b'[j] = b''[j]$, but the specific value (`0` or `1`) has not (yet) been assigned. In these cases, the IF function either selects $f'[j] = c'[j]$ and $f''[j] = c''[j]$ or selects $f'[j] = d'[j]$ and $f''[j] = d''[j]$.

- **Case 2a**: In this case, all inputs to IF are static, so $f[j]$ is static.
- **Case 2b**: $c[j]$ is static, while $d[j]$ is dynamic. In this case, the choice for $b[j]$ affects whether the bit is static or dynamic: so there are two branches here. Case 2c is similar.
- **Case 2d**: $\nabla c^{\text{in}}[j] = \nabla d^{\text{in}}[j] \in$ {`+`,`-`}. In this case $\nabla f[j] = \nabla c^{\text{in}}[j] = \nabla d^{\text{in}}[j]$ is independent of the value of $\nabla b^{\text{in}}[j]$. Thus, there is no branching resulting from this bit position.
- **Case 2c**: In this case, both $c^{\text{in}}[j]$ are $d^{\text{in}}[j]$ are dynamic, so $f[j]$ must be dynamic. However, the signs $c^{\text{in}}[j]$ are $d^{\text{in}}[j]$ differ, so the choice for $b^{\text{out}}[j]$ affects the sign of $\nabla f[j]$, resulting in two branches here. Case 2c is similar.
- **Case 2f (MSB)**: $\nabla c[j] \in$ {`+`,`-`} and $\nabla d[j] \in$ {`+`,`-`}. In this case, there is no need to specify $\nabla b[j]$ since both choices will result in $f[j]$ being dynamic, and for the MSB, the sign of the difference is irrelevant.

Case 3: $\nabla b^{\text{in}}[j] \in$ {`0`,`1`}. In these cases, the value of $\nabla b[j]$ is already specified, so the choice of input ($c[j]$ or $d[j]$) is already fixed. Hence, there is only one branch for each of these cases.

5.3 Reconstruction Infromation

In order to record the choice of static branching for all n bit positions in an efficient manner, it is sufficient to store the integer

$$\sigma = \sum_{j=0}^{n-1} BI[j] \cdot \left(\prod_{k=0}^{j-1} Br[k] \right).$$

The static branch indices are extracted from σ using the following algorithm

1. Set $j = 0$ and $s = \sigma$.
2. From $\nabla b^{\text{in}}[j]$, $\nabla c^{\text{in}}[j]$, $\nabla d^{\text{in}}[j]$, identify the corresponding case in Table 1 or Table 2, and obtain $Br[j]$.
3. Extract $BI[j] = s \pmod{Br[j]}$.
4. Compute $s = (s - BI[j])/Br[j]$.
5. Increment j and return to Step 2.

6 The Forward Search Tree

The forward search tree begins the differential path at the beginning of step 1. When searching for a forward differential, the inputs are the sequence of XOR differences in the expanded message words $\{\Delta_{\oplus} m_{i-1} = \mu_{i-1,\oplus}\}$ and the sequence of additive differences in the chaining variable (the input to the first round):

$$(\Delta_{+} a_0, \Delta_{+} b_0, \Delta_{+} c_0, \Delta_{+} d_0, \Delta_{+} e_0) = (\alpha_{0,+}, \beta_{0,+}, \gamma_{0,+}, \delta_{0,+}, \epsilon_{0,+}).$$

Since a differential may form part of a multi-block collision, the search should be able to consider cases where non-zero additive differences are already present in the chaining variable.[4] The forward search determines the possible sequence of differences $\{\Delta_+ a_i\}$; progressively determining the conditions required for each sequence.

6.1 Branches in Step 1

From Equation (1) we obtain:

$$\Delta_+ a_1 = \Delta_+ ROTL(a_0, 5) + \Delta_+ m_0 + \Delta_+ e_0 + \Delta_+ f_0.$$

Before the search can determine $\Delta_+ a_1$, the search must consider the possible additive differences $\Delta_+ ROTL(a_0, 5)$, $\Delta_+ m_0$, $\Delta_+ e_0$ and $\Delta_+ f_0$. Every time there is more than one option for $\Delta_+ ROTL(a_0, 5)$, $\Delta_+ m_0$, $\Delta_+ e_0$ or $\Delta_+ f_0$, there is a branching point and there is a choice of new subtrees to search. The following branching points can occur during Step 1:

- **Forward Dynamic Branching**. In our implementation, the various sub-branches resulting from the dynamic branching (Additive-Rotation branching, XOR-to-Additive branching, and Additive-to-XOR branching) are generated in a single component.
  - **Additive-Rotation Branching**: The search considers the possible values for $\Delta_+ ROTL(a_0, 5)$ corresponding to the specified additive difference in a_0. For each sub-branch, the values for $\nabla a_0 = \nabla b_1 = \ldots$ are restricted to the those that result in the specific value for $\Delta_+ ROTL(a_0, 5)$ associated with that sub-branch.
  - **XOR-to-Additive Branching**: The search generates a new sub-branch for each of the values for $\Delta_+ m_0$ that can be generated when the pair m_0', m_0'' has the specified XOR difference $\Delta_\oplus m_0 = \mu_{0,\oplus}$. The sign of MSB differences do not need to be specified.
  - $\Delta_+ e_0 = \epsilon_{0,+}$ is a specified input.
  - **Additive-to-XOR Branching**: The search generates a new sub-branch for each possible nabla representation ∇b_0^{in}, ∇c_0^{in}, and ∇d_0^{in} that are input to the IF function. These XOR differences must conform to the specified values for $\Delta_+ b_0^{\text{in}} = \beta_{0,+}$, $\Delta_+ c_0^{\text{in}} = \gamma_{0,+}$, and $\Delta_+ d_0^{\text{in}} = \delta_{0,+}$. The sign of each dynamic bit in the nabla representation (including the MSB) should to be specified in order to determine all possible values for Δf_0.
- **Static Branching**: The static branching processes the input nabla representations ∇b_0^{in}, ∇c_0^{in}, and ∇d_0^{in} and produces a set of branches, each corresponding to a possible value of $\Delta_+ f_0$ and some new conditions represented by ∇b_0^{out}, ∇c_0^{out}, and ∇d_0^{out}.
- **Compute** $\Delta_+ a_1$: The search can now compute $\Delta_+ a_1$ from the values of $\Delta_+ ROTL(a_0, 5)$, $\Delta_+ m_0$, $\Delta_+ e_0$ and $\Delta_+ f_0$ that have been determined for this branch.

6.2 Branches in Step 2

The following information is passed from the examination of Step 1 to the examination of Step 2: $\Delta_+ a_1$, $\Delta_+ a_0$, $\Delta_+ ROTL(a_0, 5)$, ∇b_0^{out}, ∇c_0^{out}, and ∇d_0^{out}.

- **Forward Dynamic Branching**
  - **Additive-Rotation Branching**:
    - The search considers the possible values for $\Delta_+ ROTL(a_1, 5)$ corresponding to the additive difference in a_1 determined during the branching in the previous step.
  - **XOR-to-Additive Branching**: The search generates a new sub-branch for each of the possible values for $\Delta_+ m_1$.
  - $\Delta_+ e_1 = \Delta_+ d_0$ is a specified input.
  - **Additive-to-XOR Branching**.
    - In the previous step, the search already assigned the dynamic bits and some static bits of b_0 and c_0 for this branch. Thus the dynamic bits and some static bits of $c_1^{\text{in}} = ROTL(b_0^{\text{out}}, 30)$, and $d_1^{\text{in}} = c_0^{\text{out}}$ are already specified. There is no branching associated with determining the dynamic bits of c_1^{in} and d_1^{in}.
    - Regarding, ∇b_1^{in}, recall that the Additive-Rotation branching in the previous step restricted the nabla representations $\nabla a_0 = \nabla b_1^{\text{in}}$ to those that resulted in a specific value for $\Delta_+ ROTL(a_0, 5)$. The search assigns a sub-branch for each such ∇b_1^{in}.
- **Static Branching** As for Step 1.
- The search can now compute $\Delta_+ a_2$.

6.3 Branches in Step i, $2 \leq i \leq s$.

The following information is passed from the examination of Step $(i-1)$ to the examination of Step i:

[4] Most multi-block collisions specify the additive differences in the the chaining variable. Some specify the XOR differences, but this need not be the case. For this reason, we shall assume that the XOR differences in the input to the first round are unknown.

$\Delta_+ a_{i-1}$, $\Delta_+ a_{i-2}$, $\Delta_+ ROTL(a_{i-2}, 5)$, $\nabla b_{i-2}^{\text{out}}$, $\nabla c_{i-2}^{\text{out}}$, and $\nabla d_{i-2}^{\text{out}}$.

- **Forward Dynamic Branching**
  - **Additive-Rotation Branching**:
    - The search considers the possible values for $\Delta_+ ROTL(a_{i-1}, 5)$ corresponding to the additive difference in a_{i-1} determined during the branching in the previous step.
  - **XOR-to-Additive Branching**: The search generates a new sub-branch for each of the values for $\Delta_+ m_{i-1}$.
  - $\Delta_+ e_{i-1} = \Delta_+ d_{i-2}$ is derived directly from $\nabla d_{i-2}^{\text{out}}$.
  - **Additive-to-XOR Branching**.
    - Regarding, $\nabla c_{i-1}^{\text{in}}$ and $\nabla d_{i-1}^{\text{in}}$, simply assign $\nabla c_{i-1}^{\text{in}} = ROTL(\nabla b_{i-2}^{\text{out}}, 30)$, and $\nabla d_{i-1}^{\text{in}} = \nabla c_{i-2}^{\text{out}}$..
    - Regarding, $\nabla b_{i-1}^{\text{in}}$, recall that the Additive-Rotation branching in the previous step restricted the nabla representations $\nabla a_{i-2} = \nabla b_{i-1}^{\text{in}}$ to those that resulted in a specific value for $\Delta_+ ROTL(a_{i-2}, 5)$. The search assigns a sub-branch for each such ∇b_{i-1}.
- **Static Branching** As for Step 1.
- The search can now compute $\Delta_+ a_i$.

The forward search tree consists of branches alternating between forward dynamic branching components and static branching components. When the search reaches the specified step, then the search tree ends at a leaf. At the leaf, before proceeding to the next leaf, the search outputs the resulting conditions on the values $(a_s, b_s, c_s, d_s, e_s)$ to be matched against the reverse differentials, and reconstruction information. After exhausting all sub-branches at a particular dynamic or static branching, the search returns down (towards the root of the tree) to the previous branching, and takes the next sub-branch from there.

7 The Reverse Search Tree

Like the forward search tree, the reverse search tree consists of branches alternating between reverse dynamic branching components and static branching components. The reverse search tree is somewhat simpler than the forward search tree since there is no need to consider addition-rotation branching for e_{i-1}. Note that we consider the value of $d_{i-2} = e_{i-1}$ as an input to the IF function in round $(i-1)$ before we consider $ROTL(a_{i-5}, 5) = ROTL(e_{i-1}, 7)$, and to determine the static branches we must have already fully specified the XOR bit differences.

With the reverse search tree, we begin the differential at the end of step 20. When searching for a reverse differential, the inputs are the sequence of XOR differences in the expanded message words $\{\Delta_\oplus m_{i-} = \mu_{i-1,\oplus}\}$ and the sequence of differences in the the input to the second round. The nature of the intended differential path through Round 2 (steps 21 to 40) means that conditions on $(a_{20}, b_{20}, c_{20}, d_{20}, e_{20})$ are typically XOR differences, so the input conditions for $(a_{20}, b_{20}, c_{20}, d_{20}, e_{20})$ are of the form

$$(\Delta_\oplus a_{20}, \Delta_\oplus b_{20}, \Delta_\oplus c_{20}, \Delta_\oplus d_{20}, \Delta_\oplus e_{20}) = (\alpha_{20,\oplus}, \beta_{20,\oplus}, \gamma_{20,\oplus}, \delta_{20,\oplus}, \epsilon_{20,\oplus}).$$

Note that $b_{20} = a_{19}$, $c_{20} = ROTL(a_{18}, 5)$, $d_{20} = ROTL(a_{17}, 5)$, and $e_{20} = ROTL(a_{16}, 5)$. Thus, our "initial conditions" already specify a significant number of the conditions required in rounds 16 to 20.

7.1 Step 20

Consider reversing step 20:

$$\Delta_+ e_{19} = \Delta_+ a_{20} - \Delta_+ ROTL(a_{19}, 5) - \Delta_+ m_{19} - \Delta_+ f_{19}.$$

Before the search can determine $\Delta_+ e_{19}$, the search must consider the possible additive differences $\Delta_+ a_{20}$, $\Delta_+ ROTL(a_{19}, 5)$, $\Delta_+ m_{19}$ and $\Delta_+ f_{19}$.

- **Reverse Dynamic Branching**. This first step considered in the reverse search consists mainly of XOR-to-Additive branches, since the initial conditions are termed exclusively in terms of XOR differences.
  - **XOR-to-Additive Branching**:
    - The search considers the possible values for $\Delta_+ a_{20}$ correspond to the specified XOR difference $\Delta_\oplus a_{20} = \alpha_{20,\oplus}$. This will result in specifying all dynamic bits of a_{20} (MSB excepted).
    - The search considers the possible values for $\Delta_+ ROTL(a_{19}, 5)$ corresponding to the specified XOR difference in $\Delta_\oplus b_{20} = \beta_{20,\oplus}$.
    - The search generates a new sub-branch for each of the possible values for $\Delta_+ m_{19}$.
    - The search generates a new sub-branch for each possible nabla representation $\nabla b_{19}^{\text{in}}$, $\nabla c_{19}^{\text{in}}$, and $\nabla d_{19}^{\text{in}}$. These additive differences must conform to the specified values for $\Delta_\oplus b_{19} = \Delta_\oplus c_{20} = \gamma_{20,\oplus}$, $\Delta_\oplus c_{19} = \Delta_\oplus d_{20} = \delta_{20,\oplus}$, and $\Delta_\oplus d_{19} = \Delta_\oplus e_{20} = \epsilon_{20,\oplus}$ given in the input conditions. The sign of each dynamic bit in the

nabla representation (including the MSB) should to be specified in order to determine all possible values for Δf_{19}.

- **Static Branching**: As for the forward search , the static branching processes the input nabla representations $\nabla b_{19}^{\text{in}}$, $\nabla c_{19}^{\text{in}}$, and $\nabla d_{19}^{\text{in}}$ and produces a set of branches, each corresponding to a possible value of $\Delta_+ f_{19}$ and some new conditions represented by $\nabla b_{19}^{\text{out}}$, $\nabla c_{19}^{\text{out}}$, and $\nabla d_{19}^{\text{out}}$.
- The search can now compute $\Delta_+ e_{19}$ from the values of $\Delta_+ a_{20}$, $\Delta_+ ROTL(a_{19}, 5)$, $\Delta_+ m_{19}$, and $\Delta_+ f_{19}$ that have been determined for this branch.

7.2 Step 19

Each combination of these branches has the potential to result in a distinct $\Delta_+ e_{19}$. Now, let us look at the next step down (Step 19):

$$\begin{aligned}\Delta_+ e_{18} = \Delta_+ a_{19} - \Delta_+ ROTL(a_{18}, 5) \\ - \Delta_+ m_{18} - \Delta_+ f_{18}\end{aligned}$$

- **Reverse Dynamic Branching**
  - **XOR-to-Additive Branching** Recall that $\Delta_\oplus a_{19} = \Delta_\oplus b_{20} = \beta_{20,\oplus}$ was specified in the initial conditions. In order to specify $\Delta_+ ROTL(a_{19}, 5)$ in step 20,the search specifies the signs of the all dynamic bits of a_{19}, with the exception of bit 27 (if bit 27 is dynamic: that is, if $\beta_{20,\oplus}[27] = 1$). Thus, $\beta_{20,\oplus}[27] = 1$, then the search is required to consider the two branches corresponding to $\nabla a_{19}[27] =$ `+`, and $\nabla a_{19}[27] =$ `-`. Otherwise, there is only one branch.
  - Since we fully specify the dynamic bits of $b_{19} = a_{18}$ in analysing step 20, this will fully define $\Delta_+ ROTL(a_{18}, 5)$.
  - The search generates a new sub-branch for each of the possible values for $\Delta_+ m_{18}$.
  - **Additive-to-XOR Branching** The search considers the possible values for $\nabla d_{18}^{\text{in}}$ corresponding to the additive difference $\Delta_+ d_{18} = \Delta_+ e_{19}$ determined after the branching in step 20.
  - Assign $\nabla b_{18}^{\text{in}} = ROTL(\nabla c_{19}^{\text{out}}, 2)$, and $\nabla c_{18}^{\text{in}} = \nabla d_{19}^{\text{out}}$.
- **Static Branching**: As for forward search.
- The search can now compute $\Delta_+ e_{18}$.

7.3 Step i, $i \leq 18$

We now consider the general case for Step i, $i \leq 18$:

$$\begin{aligned}\Delta_+ e_{i-1} = \Delta_+ a_i - \Delta_+ ROTL(a_{i-1}, 5) \\ - \Delta_+ m_{i-1} - \Delta_+ f_{i-1}.\end{aligned}$$

- **Reverse Dynamic Branching**
  - $\Delta_+ a_i$ is fully specified by $\nabla b_{i+1}^{\text{out}}$.
  - $\Delta_+ ROTL(a_{i-1}, 5)$ is fully specified by ∇b_i^{out}.
  - The search generates a new sub-branch for each of the possible values for $\Delta_+ m_{i-1}$.
  - **Additive-to-XOR Branching** The search considers the possible values for $\nabla d_{i-1}^{\text{in}}$ corresponding to the additive difference $\Delta_+ d_{i-1} = \Delta_+ e_i$ determined after the branching in step $(i+1)$.
  - Assign $\nabla b_{i-1}^{\text{in}} = ROTL(\nabla c_i^{\text{out}}, 2)$, and $\nabla c_{i-1}^{\text{in}} = \nabla d_i^{\text{out}}$.
- **Static Branching**: As for forward search.
- The search can now compute $\Delta_+ e_{i-1}$.

8 Matching Forward and Reverse Differential Paths

Now that we can create forward and reverse paths, the next piece in the puzzle is to determine when a forward path and reverse path match up.

A forward path ends with the following conditions on the resulting state:

- $\Delta_+ a_s = \alpha_+^{(f)}$, $\Delta_+ b_s = \beta_+^{(f)}$;
- $\nabla c_s \hookrightarrow \gamma^{(f)}$, $\nabla d_s \hookrightarrow \delta^{(f)}$, $\nabla e_s \hookrightarrow \epsilon^{(f)}$;
- Arrays $\texttt{v0}^{(f)}\texttt{[]}$ and $\texttt{v1}^{(f)}\texttt{[]}$.

A reverse path ends with the following conditions on the resulting state:

- $\nabla a_s \hookrightarrow \alpha^{(r)}$, $\nabla b_s \hookrightarrow \beta^{(r)}$, $\nabla c_s \hookrightarrow \gamma^{(r)}$, $\nabla d_s \hookrightarrow \delta^{(r)}$;
- $\Delta_+ e_s = \epsilon_+^{(r)}$;
- Arrays $\texttt{v0}^{(r)}\texttt{[]}$ and $\texttt{v1}^{(r)}\texttt{[]}$.

In order for a forward path and reverse path to form a full 20-step path from $(a_0, b_0, c_0, d_0, e_0)$ to $(a_{20}, b_{20}, c_{20}, d_{20}, e_{20})$, the conditions on $(a_s, b_s, c_s, d_s, e_s)$ required by both the forward path and reverse path must be satisfied. If the conditions on $(a_s, b_s, c_s, d_s, e_s)$ required by both the forward path and reverse path can be satisfied, then we will say that the two paths are *compatible*.

8.1 Integrating Forward and Reverse Paths

Table 3 shows how compatible forward and reverse paths interact. The columns headed by 'f' refer to conditions related to the forward path, which columns headed by 'r' refer to conditions related to the reverse path. The conditions on each internal variable falls into one of the following categories:

– (f,r) = $(\nabla, \cdot)$: (that is, the 'f' column contains a ∇ and the 'r' column contains a '$\cdot$'). The forward path imposes "nabla" conditions (XOR and additive differences) and the reverse path imposes no conditions.
– (f,r) = $(\nabla, +)$: The forward path imposes nabla conditions and the reverse path must agree with the additive differences.
– (f,r) = (∇, ∇): Both the forward path and reverse path impose nabla conditions.
– (f,r) = $(+, \nabla)$: The reverse path imposes nabla conditions and the forward path must agree with the additive differences.
– (f,r) = $(\cdot, \nabla)$. The reverse path imposes nabla conditions, and the forward path none.

Step	a_i		b_i		c_i		d_i		e_i	
i	f	r	f	r	f	r	f	r	f	r
$s-5$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$
$s-4$	∇	$+$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$
$s-3$	∇	∇	∇	$+$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$
$s-2$	∇	∇	∇	∇	∇	$+$	∇	$\cdot$	∇	$\cdot$
$s-1$	$+$	∇	∇	∇	∇	∇	∇	$+$	∇	$\cdot$
s	$+$	∇	$+$	∇	∇	∇	∇	∇	∇	$+$
$s+1$	$\cdot$	∇	$+$	∇	$+$	∇	∇	∇	∇	∇
$s+2$	$\cdot$	∇	$\cdot$	∇	$+$	∇	$+$	∇	∇	∇
$s+3$	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$+$	∇	$+$	∇
$s+4$	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$+$	∇
$s+5$	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇	$\cdot$	∇

Table 3. Integrating conditions from a forward path and compatible reverse path. Only steps $(s-5)$ to $(s+5)$ are shown. See Section 8.1 for notation.

8.2 Compatibility Conditions

Requirements for a_s, b_s and e_s. The forward path imposes the more flexible conditions of the form $\Delta_+ a_s = \alpha_+^{(f)}$. For the conditions on a_s to be satisfied by both paths, it is sufficient to confirm that $\alpha^{(r)}$ corresponds to an additive difference equal to $\alpha_+^{(f)}$. Consequently, it is sufficient for the reverse search to store $\alpha_+^{(r)}$ rather than the whole nabla representation $\alpha^{(r)}$. Similarly, the reverse search need only store $\beta_+^{(r)}$ to compare with $\beta_+^{(f)}$, and the forward search need only store $\epsilon_+^{(f)}$ to compare with $\epsilon_+^{(r)}$.

Requirements for Variables c_s and d_s. The forward path imposes the conditions $\nabla c_s \hookrightarrow \gamma^{(f)}$, $\nabla d_s \hookrightarrow \delta^{(f)}$, and the reverse path imposes the conditions $\nabla c_s \hookrightarrow \gamma^{(r)}$, $\nabla d_s \hookrightarrow \delta^{(r)}$. The first few requirements for compatibility are:

– *Rule 1:* The static and dynamic bits of $\gamma^{(f)}$ and $\gamma^{(r)}$ must occur in the same positions; and
– *Rule 2:* The static and dynamic bits of $\delta^{(f)}$ and $\delta^{(r)}$ must occur in the same positions.
– *Rule 3:* The paths are not compatible if $(\gamma^{(f)}[j], \gamma^{(r)}[j]) \in \{(+,-),(-,+)\}$ for some j.
– *Rule 4:* The paths are not compatible if $(\delta^{(f)}[j], \delta^{(r)}[j]) \in \{(+,-),(-,+)\}$ for some j.

There are not additional tests required for dynamic bit positions. It remains to describe when conditions on static bits allow compatible paths. The main obstacle to this is the relational conditions. These correspond to bit positions contains a pointer to an element of the `v0` or `v1` arrays.

In many cases, the conditions are further refined by in the arrays are refined assigning specific values ("`0`" or "`1`") to elements of the `v0` or `v1` arrays. Such values can be simply copied to the corresponding positions in the nabla representations. Where the conditions are not further refined, we simply assign "`*`" to the corresponding positions in the nabla representations: and address the relational conditions independently as required. Let $\overline{\gamma}^{(f)}$, $\overline{\gamma}^{(r)}$, $\overline{\delta}^{(f)}$, $\overline{\delta}^{(r)}$ denote the nabla representations resulting after $\gamma^{(f)}$, $\gamma^{(r)}$, $\delta^{(f)}$, $\delta^{(r)}$, have the values "`*`", "`0`" or "`1`" substituted (as appropriate) for pointers to `v0` or `v1` array.

– *Rule 5:* The paths are not compatible if $(\overline{\gamma}^{(f)}[j], \overline{\gamma}^{(r)}[j]) \in \{(0,1),(1,0)\}$ for some j.
– *Rule 6:* The paths are not compatible if $(\overline{\delta}^{(f)}[j], \overline{\delta}^{(r)}[j]) \in \{(0,1),(1,0)\}$ for some j.

8.3 Relational Conditions

The forward path imposes no static conditions involving a_s and b_s, so any relational conditions imposed by the reverse path can always be satisfied by the forward path. Similarly, relational conditions imposed by the forward path on e_s can always be satisfied by the reverse path. The only relational conditions that can cause conflicts are of the form $c_s[j] = d_s[j]$ or $c_s[j] \neq d_s[j]$. Such conditions can only be imposed by the differential through Step s in the reverse path. We need only ensure that these conditions can be satisfied by the values in the corresponding nabla representations for the forward differential path.

– *Rule 7:* If the reverse path imposes a condition $c_s[j] = d_s[j]$ during step s, then the paths are not compatible if $(\overline{\gamma}^{(f)}[j], \overline{\delta}^{(f)}[j]) \in \{(0,1),(1,0)\}$.
– *Rule 8:* If the reverse path imposes a condition $c_s[n-1] \neq d_s[n-1]$ during step s, then the paths are not compatible if $(\overline{\gamma}^{(f)}[j], \overline{\delta}^{(f)}[j]) \in \{(0,0),(1,1)\}$.

9 Conclusion

The differential paths are formed by searching the forward search tree (Section 6) and reverse search tree (Section 7), generated by XOR-to-Additive Branching, Additive-to-XOR Branching (Section 4), Additive-Rotation Branching, and Static Branching (Section 5). The forward search and reverse search have been successfully implemented. We have determined the criteria to use in matching forward differential to a reverse differential, but we are only just ready to begin implementing this. After implementing the matching criteria, the next obstacle will be finding an optimal order for searching through the search tree.

References

1. E. Biham and R. Chen, *Near-Collisions of SHA-0* Advances in Cryptology-CRYPTO 2004, Lecture Notes in Computer Science, vol.3152, M. Franklin (Ed.), pp. 290-305, Springer-Verlag, 2004.
2. E. Biham and R. Chen, *New results on SHA-0 and SHA-1* Short talk presented at CRYPTO 2004 Rump Session, 2004.
3. E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby. *Collisions of SHA-0 and Reduced SHA-1,* Advances in Cryptology-EUROCRYPT 2005, Lecture Notes in Computer Science, vol.3494, R. Crmaer (Ed.), pp. 36-57, Springer-Verlag, 2005.
4. F. Chabaud and A. Joux, *Differential Collisions in SHA-0,* Advances in Cryptology-CRYPTO'98, Lecture Notes in Computer Science, vol.1462, R. Cramer (Ed.), pp.56-71, Springer-Verlag, 1998.
5. National Institute of Standards and Technology, *Federal Information Processing Standards (FIPS) Publication 180-2, Secure Hash Standard (SHS),* February, 2004.
6. H. Gilbert and H. Hanschuh, *Security Analysis of SHA-256 and sisters,* Selected Areas in Cryptography, SAC 2003, Ottawa, Canada, Lecture Notes in Computer Science, vol. 3006, M. Matsui and R. Zuccheratopp (Eds), pp. 175-193, Springer, 2004.
7. A. Joux, *Collisions in SHA-0* Short talk presented at CRYPTO 2004 Rump Session, 2004.
8. C. Jutla and A. Patthak, *A Matching Lower Bound on the Minimum Weight of SHA-1 Expansion Code,* Cryptology ePrint Archive, Report 2005/266, 2005. See `http://eprint.iacr.org/`
9. V. Klima, *Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications,* Cryptology ePrint Archive, Report 2005/102, 2005. See `http://eprint.iacr.org/`
10. V. Klima, *Tunnels in Hash Functions: MD5 Collisions Within a Minute,* Cryptology ePrint Archive, Report 2006/105, 2006. See `http://eprint.iacr.org/`
11. J. Liang and X. Lai, *Improved Collision Attack on Hash Function MD5,* Cryptology ePrint Archive, Report 2005/425, 2005. See `http://eprint.iacr.org/`
12. K. Matusiewicz and J. Pieprzyk, *Finding good differential patterns for attacks on SHA-1,* Cryptology ePrint Archive, Report 2004/364, 2004. See `http://eprint.iacr.org/`
13. 0. Mikle, *Practical Attacks on Digital Signatures Using MD5 Message Digest,* Cryptology ePrint Archive, Report 2004/356 , 2004. See `http://eprint.iacr.org/`
14. A. Menezes, P van Oorschot and S. Vanstone, *Handbook of Applied Cryptography,* CRC Press LLC, 1997.
15. V. Rijmen and E. Oswald, *Update on SHA-1,* Cryptology ePrint Archive, Report 2005/010, 2005. See `http://eprint.iacr.org/`
16. R. Rivest, *The MD5 Message-Digest Algorithm,* Internet RFC 1321, April 1992.
17. Y. Sasaki, Y. Naito, N. Kunihiro and K. Ohta, *Improved Collision Attack on MD5,* Cryptology ePrint Archive, Report 2005/400, 2005. See `http://eprint.iacr.org/`
18. Y. Sasaki, Y. Naito, J. Yajima, T. Shimoyama, N. Kunihiro and K. Ohta, *How to Construct Sufficient Condition in Searching Collisions of MD5,* Cryptology ePrint Archive, Report 2006/074, 2006. See `http://eprint.iacr.org/`
19. M. Stevens, *Fast Collision Attack on MD5,* Cryptology ePrint Archive, Cryptology ePrint Archive, Report 2006/104, 2006. See `http://eprint.iacr.org/`
20. M. Sugita, M. Kawazoe and H. Imai, *Gröbner Basis Based Cryptanalysis of SHA-1,* Cryptology ePrint Archive, Report 2006/098, 2006. See `http://eprint.iacr.org/`
21. X. Wang, D. Feng, X. Lai and H. Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD,* Cryptology ePrint Archive, Report 2004/199, 2004. See `http://eprint.iacr.org/`
22. X. Wang, X. Lai, D. Feng, H. Chen and H. Yu, *How to break MD5 and other Hash Functions,* Advances in Cryptology-EUROCRYPT 2005, Lecture Notes in Computer Science, vol.3494, pp 19-35, R. Cramer (Ed.), Springer-Verlag, 2005.
23. X. Wang, H. Yu and Y. Yin, *Efficient Collision Search Attacks on SHA-0,* Advances in Cryptology-CRYPTO 2005, Lecture Notes in Computer Science, vol.3621, pp.1-16, V. Shoup (Ed.), Springer-Verlag, 2005.
24. X. Wang, Y. Yin and H. Yu, *Finding Collisions in the Full SHA-1,* Advances in Cryptology-CRYPTO 2005, Lecture Notes in Computer Science, vol.3621, pp.17-36, V. Shoup (Ed.), Springer-Verlag, 2005.
25. J. Yajima and T. Shimoyama, *Wang's sufficient conditions of MD5 are not sufficient,* Cryptology ePrint Archive, Report 2005/263, 2005. See `http://eprint.iacr.org/`