Practical Free-Start Collision Attacks
on 76-step `SHA-1`

Pierre Karpman[1,2][*], Thomas Peyrin[2][**], and Marc Stevens[3]

[1] Inria, France
[2] Nanyang Technological University, Singapore
[3] Centrum Wiskunde & Informatica, The Netherlands

pierre.karpman@inria.fr, thomas.peyrin@ntu.edu.sg, marc.stevens@cwi.nl

Abstract. In this paper we analyze the security of the compression function of `SHA-1` against collision attacks, or equivalently free-start collisions on the hash function. While a lot of work has been dedicated to the analysis of `SHA-1` in the past decade, this is the first time that free-start collisions have been considered for this function. We exploit the additional freedom provided by this model by using a new start-from-the-middle approach in combination with improvements on the cryptanalysis tools that have been developed for `SHA-1` in the recent years. This results in particular in better differential paths than the ones used for hash function collisions so far. Overall, our attack requires about 2^{50} evaluations of the compression function in order to compute a one-block free-start collision for a 76-step reduced version, which is so far the highest number of steps reached for a collision on the `SHA-1` compression function. We have developed an efficient GPU framework for the highly branching code typical of a cryptanalytic collision attack and used it in an optimized implementation of our attack on recent GTX 970 GPUs. We report that a single cheap US$ 350 GTX 970 is sufficient to find the collision in less than 5 days. This showcases how recent mainstream GPUs seem to be a good platform for expensive and even highly-branching cryptanalysis computations. Finally, our work should be taken as a reminder that cryptanalysis on `SHA-1` continues to improve. This is yet another proof that the industry should quickly move away from using this function.

Keywords: `SHA-1`, hash function, cryptanalysis, free-start collision, GPU implementation.

1 Introduction

Cryptographic hash functions are essential components in countless security systems for very diverse applications. Informally, a hash function H is a function that takes an arbitrarily long message M as input and outputs a fixed-length hash value of size n bits. One of the main security requirements for a cryptographic hash function is to be collision resistant: it should be hard for an adversary to find two distinct messages M, $\hat{M}$ leading to the same hash value $H(M) = H(\hat{M})$ in less than $2^{\frac{n}{2}}$ calls to H. Most standardized hash functions are based on the Merkle-Damgård paradigm [27,6] which iterates a compression function h that updates a fixed-size internal state (also called chaining value) with fixed-size message blocks. This construction allows a simple and very useful security reduction: if the compression function is collision-resistant, then so is the corresponding hash function. Since the compression function has two inputs, an attacker may use this extra freedom to mount attacks that are not possible on the complete hash function; on the other hand, one loses the ability to chain message blocks. We can then distinguish between two classical attack models: a *free-start* collision is a pair of different message and chaining value (c, m), $(\hat{c}, \hat{m})$ leading to a collision after applying h: $h(c, m) = h(\hat{c}, \hat{m})$. A *semi-free-start* collision works similarly, with the additional restriction that the chaining values c and $\hat{c}$ must be equal. It is important to note that the Merkle-Damgård security reduction assumes that any type of collision for the compression function should be intractable for an attacker, including free-start collisions.

The most famous and probably most used hash function as of today is SHA-1 [29]. This function belongs to the MD-SHA family, that originated with MD4 [34]. Soon after its publication, MD4 was believed to be insufficiently secure [9] and a practical collision was later found [11]. Its improved version, MD5 [35], was widely deployed in countless applications, even though collision attacks on the compression function were quickly identified [10]. The function was completely broken and collisions were found in the groundbreaking work from Wang *et al.* [43]. A more powerful type of collision attack called *chosen-prefix collision attack* against MD5 was later introduced by Stevens *et al.* [40]. Irrefutable proof that hash function collisions indeed form a realistic and significant threat to Internet security was then provided by Stevens *et al.* [41] with their construction of a Rogue Certification Authority that in principle completely undermined HTTPS security. This illustrated further that the industry should move away from weak cryptographic hash functions and should not wait until cryptanalytic advances actually prove to be a direct threat to security. Note that one can use *counter-cryptanalysis* [38] to protect against such digital signature forgeries during the strongly advised migration away from MD5 and SHA-1.

Before the impressive attacks on MD5, the NIST had standardized the hash function SHA-0 [28], designed by the NSA and very similar to MD5. This function was quickly very slightly modified and became SHA-1 [29], with no justification provided. A plausible explanation came from the pioneering work of Chabaud and Joux [4] who found a theoretical collision attack that applies to SHA-0 but not to SHA-1. Many improvements of this attack were subsequently proposed [1]

and an explicit collision for SHA-0 was eventually computed [2]. However, even though SHA-0 was practically broken, SHA-1 remained free of attacks until the work of Wang *et al.* [42] in 2005, who gave the very first theoretical collision attack on SHA-1 with an expected cost equivalent to 2^{69} calls to the compression function. This attack has later been improved several times, the most recent improvement being due to Stevens [39], who gave an attack with estimated cost 2^{61}; yet no explicit collision has been computed so far. With the attacks on the full SHA-1 remaining impractical, the community focused on computing collisions for reduced versions: 64 steps [8] (with a cost of 2^{35} SHA-1 calls), 70 steps [7] (cost 2^{44} SHA-1), 73 steps [13] (cost $2^{50.7}$ SHA-1) and the latest advances reached 75 steps [14] (cost $2^{57.7}$ SHA-1) using extensive GPU computation power. As of today, one is advised to use *e.g.* SHA-2 [30] or the hash functions of the future SHA-3 standard [31] when secure hashing is needed.

In general, two main points are crucial when dealing with a collision search for a member of the MD-SHA family of hash functions (and more generally for almost every hash function): the quality of the differential paths used in the attack and the amount and utilization of the remaining freedom degrees. Regarding SHA-0 or SHA-1, the differential paths were originally built by linearizing the step function and by inserting small perturbations and corresponding corrections to avoid the propagation of any difference. These so-called local collisions [4] fit nicely with the linear message expansion of SHA-0 and SHA-1 and made it easy to generate differential paths and evaluate their quality. However, these linear paths have limitations since not so many different paths can be used as they have to fulfill some constraints (for example no difference may be introduced in the input or the output chaining value). In order to relax some of these constraints, Biham *et al.* [2] proposed to use several successive SHA-1 compression function calls to eventually reach a collision. Then, Wang *et al.* [42] completely removed these constraints by using only two blocks and by allowing some part of the differential paths to behave non-linearly (*i.e.* not according to a linear behavior of the SHA-1 step function). Since the non-linear parts have a much lower differential probability than the linear parts, to minimize the impact on the final complexity they may only be used where freedom degrees are available, that is during the first steps of the compression function. Finding these non-linear parts can in itself be quite hard, and it is remarkable that the first ones were found by hand. Thankfully, to ease the work of the cryptanalysts, generating such non-linear parts can now be done automatically, for instance using the guess-and-determine approach of De Cannière and Rechberger [8], or the meet-in-the-middle approach of Stevens *et al.* [37,15]. In addition, joint local collision analysis [39] for the linear part made heuristic analyzes unnecessary and allows to generate optimal differential paths.

Once a differential path has been chosen, the remaining crucial part is the use of the available freedom degrees when searching for the collision. Several techniques have been introduced to do so. First, Chabaud and Joux [4] noticed that in general the 15 first steps of the differential path can be satisfied for free since the attacker can fix the first 16 message words independently, and thus

fulfill these steps one by one. Then, Biham and Chen [1] introduced the notion of neutral bits, that allows the attacker to save conditions for a few additional steps. The technique is simple: when a candidate following the differential path until step $x > 15$ is found, one can amortize the cost for finding this valid candidate by generating many more almost for free. Neutral bits are small modifications in the message that are very likely not to invalidate conditions already fulfilled in the x first steps. In opposition to neutral bits, the aim of message modifications [42] is not to multiply valid candidates but to correct the wrong ones: the idea is to make a very specific modification in a message word, so that a condition not verified at a later step eventually becomes valid with very good probability, but without interfering with previously satisfied conditions. Finally, one can cite the tunnel technique from Klíma [20] and the auxiliary paths (or boomerangs) from Joux and Peyrin [18], that basically consist in pre-set, but more powerful neutral bits. Which technique to use, and where and how to use it are complex questions for the attacker and the solution usually greatly depends on the specific case that is being analyzed.

Our contributions. In this paper, we study the free-start collision security of `SHA-1`. We explain why a start-from-the-middle approach can improve the current best collision attacks on the `SHA-1` compression function regarding two keys points: the quality of the differential paths generated, but also the amount of freedom degrees and the various ways to use them. Furthermore, we present improvements to derive differential paths optimized for collision attacks using an extension of joint local-collision analysis. All these improvements allow us to derive a one-block free-start collision attack on 76-step `SHA-1` for a rather small complexity equivalent to about 2^{50} calls to the primitive. We have fully implemented the attack and give an example of a collision in the full version of the paper [19].

We also describe a GPU framework for a very efficient GPU implementation of our attack. The computation complexity is quite small as a single cheap US\$ 350 GTX 970 can find a collision in less than 5 days, and our cheap US\$ 3000 server with four GTX 970 GPUs can find one in slightly more than one day on average. In comparison, the 75-step collision from [14] was computed on the most powerful supercomputer in Russia at the time, taking 1.5 month on 455 GPUs on average. This demonstrates how recent mainstream GPUs can easily be used to perform big cryptanalysis computations, even for the highly branching code used in cryptanalytic collision attacks. Notably, our approach leads to a very efficient implementation where a single GTX 970 is equivalent to about 140 recent high-clocked Haswell cores, whereas the previous work of Grechnikov and Adinetz estimates an Nvidia Fermi GPU to be worth 39 CPU cores [14].

Moreover, we emphasize that we have found the collision that has reached the highest number of `SHA-1` compression function steps as of today. Finally, this work serves as a reminder that cryptanalysis on `SHA-1` continues to improve and that industry should now quickly move away from using this primitive.

Outline. In Section 2 we first describe the SHA-1 hash function. In Section 3 we describe the start-from-the-middle approach and how it can provide an improvement when looking for (semi)-free-start collisions on a hash function. We then study the case of 76-step reduced SHA-1 with high-level explanations of the application of the start-from-the-middle approach, the differential paths, and the GPU implementation of the attack in Section 4. The reader interested by more low-level details can then refer to Section 5. Finally, we summarize our results in Section 6.

2   The SHA-1 hash function

We give here a short description of the SHA-1 hash function and refer to [29] for a more exhaustive treatment. SHA-1 is a 160-bit hash function belonging to the MD-SHA family. Like many hash functions, SHA-1 uses the Merkle-Damgård paradigm [6,27]: after a padding process, the message is divided into k blocks of 512 bits each. At every iteration of the compression function h, a 160-bit chaining value cv_i is updated using one message block m_{i+1}, *i.e.* $cv_{i+1} = h(cv_i, m_{i+1})$. The initial value $IV = cv_0$ is a predefined constant and cv_k is the output of the hash function.

As for most members of the MD-SHA family, the compression function h uses a block cipher Enc in a Davies-Meyer construction: $cv_{i+1} = Enc(m_{i+1}, cv_i) + cv_i$, where $Enc(x, y)$ denotes the encryption of plaintext y with key x. The block cipher itself is an 80-step (4 rounds of 20 steps each) generalized Feistel network which internal state is composed of five branches (or internal registers) $(A_i, B_i, C_i, D_i, E_i)$ of 32-bit each. At each step, a 32-bit extended message word W_i is used to update the five internal registers:

$$\begin{cases} A_{i+1} = (A_i \lll 5) + f_i(B_i, C_i, D_i) + E_i + K_i + W_i, \\ B_{i+1} = A_i, \\ C_{i+1} = B_i \ggg 2, \\ D_{i+1} = C_i, \\ E_{i+1} = D_i. \end{cases}$$

where K_i are predetermined constants and f_i are Boolean functions defined in Table 2-1. Note that all updated registers but A_{i+1} are just rotated copies of another register, so one can only consider the register A at each iteration. Thus, we can simplify the step function as:

$$A_{i+1} = (A_i \lll 5) + f_i(A_{i-1}, A_{i-2} \ggg 2, A_{i-3} \ggg 2) + (A_{i-4} \ggg 2) + K_i + W_i.$$

Finally, the extended message words W_i are computed from the 512-bit message block, which is split into 16 32-bit words $M_0, \ldots, M_{15}$. These 16 words are then expanded linearly into the 80 32-bit words W_i as follows:

$$W_i = \begin{cases} M_i, & \text{for } 0 \le i \le 15 \\ (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, & \text{for } 16 \le i \le 79 \end{cases}$$

Table 2-1. Boolean functions and constants of `SHA-1`

round	step i	$f_i(B, C, D)$	K_i
1	$0 \leq i < 20$	$f_{IF} = (B \wedge C) \oplus (\overline{B} \wedge D)$	`0x5a827999`
2	$20 \leq i < 40$	$f_{XOR} = B \oplus C \oplus D$	`0x6ed6eba1`
3	$40 \leq i < 60$	$f_{MAJ} = (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	`0x8fabbcdc`
4	$60 \leq i < 80$	$f_{XOR} = B \oplus C \oplus D$	`0xca62c1d6`

For the sake of completeness, since our attacks compute the internal cipher *Enc* both in the forward (encryption) and backward (decryption) directions, we also give below the description of the inverse of the state update function:

$$A_i = (A_{i+5} - W_{i+4} - K_{i+4} - f_{i+4}(A_{i+3}, A_{i+2} \ggg 2, A_{i+1} \ggg 2) - (A_{i+4} \lll 5)) \lll 2$$

and of the message expansion: $W_i = (W_{i+16} \ggg 1) \oplus W_{i+13} \oplus W_{i+8} \oplus W_{i+2}$.

3 A start-from-the-middle approach

The first example of an attack starting from the middle of a hash function is due to Dobbertin [11], who used it to compute a collision on `MD4`. Start-from-the-middle methods are also an efficient approach to obtain distinguishers on hash functions, such as Saarinen's slide distinguisher on the `SHA-1` compression function [36]. Rebound attacks [25] and their improvements [24,12,21,16] can be considered as a start-from-the-middle strategy tailored to the specific case of `AES`-like primitives. Start-from-the-middle has also been used to improve the analysis of functions based on parallel branches, such as `RIPEMD-128` [22]. All these attacks leverage the fact that in some specific scenarios, starting from the middle may lead to a better use of the freedom degrees available.

In general, a start-from-the-middle approach for collision search leads to a free-start or semi-free-start attack. Indeed, since one might not use the freedom degrees in the first steps of the compression function anymore, it is harder to ensure that the initial state (*i.e.* the chaining value) computed from the middle is equal to the specific initial value of the function's specifications. However, if the starting point is not located too far from the beginning, one may still be able to do so; this is for example the case of the recent attack on Grøstl [26]. In this work, we focus on the search of free-start collisions and consider that the IV can be chosen by the attacker. For `SHA-1`, this adds 160 bits of freedom that can be set in order to fulfill conditions to follow a differential path.

Furthermore, in the case of `SHA-1`, one can hope to exploit a start-from-the-middle approach even more to improve previous works in two different ways. Firstly, the set of possible differential paths to consider increases. Secondly, the freedom degrees now operate in two directions: forward and backward.

More choice for the differential paths. The linear differential paths (generated from the mask of introduced perturbations, also called *disturbance vector* or DV) used in all existing collision attacks on `SHA-1` can be concisely described in two families [23]. Both types have the feature that the complexity of following the path is unevenly distributed along the 80 steps of `SHA-1` (this remains true for attacks on reduced versions as well). Furthermore, because a typical attack replaces a portion of the linear differential path by a non-linear part, this latter one defines a series of steps where the complexity of the linear path is basically irrelevant. In the case of a start-from-the-middle attack, one can choose where to locate this non-linear part, and thereby gains more flexibility in the choice of the linear path to use.

Two-way use of the freedom degrees. In a start-from-the-middle setting, one freely chooses an initial state in the middle of the function instead of necessarily starting from the IV in the beginning. Therefore, the differential path conditions may be related both to *forward* and *backward* computations from the middle state, and the same goes for the freedom available in the first 16 words of the message. Because one now has more possibilities to exploit them, one can hope to make a better use of these freedom degrees. For example, we can imagine two-way neutral bits, that is applying neutral bits in both forward and backward directions. This would potentially allow the attacker to obtain a few free steps not only in the forward direction as in previous works, but also in the backward direction. Of course, one must be careful about the non-independence between the forward and backward computations. Obviously, the same reasoning can apply to other freedom degrees utilization techniques such as message modifications, tunnels or boomerangs.

In the next two sections, we detail how we applied this approach to the search of free-start collisions for `SHA-1`.

4 A high-level view of the `SHA-1` free-start collision attack

4.1 Start-from-the-middle

There is one main choice that must be made when using a start-from-the-middle approach for an attack on `SHA-1`, that is which consecutive 16 steps are used to apply advanced message modification techniques or neutral bits; in our terminology the offset corresponding to this choice is called the *main block offset*. Any simple change in those 16 steps propagates to all other steps, in particular differences propagate backwards from these 16 steps down to step 0 and thereby affect the input chaining values. Note that for regular attacks on `SHA-1`, the main block offset must be 0 to ensure that the chaining value is never altered.

For our neutral bits, we found that using a main block offset of 6 was optimal. Therefore neutral bits are applied on the 16 message words $W_{6\ldots21}$ and a neutral bit in W_i affects steps $5, 4, \ldots, 0$ backwards and steps $i, i+1, \ldots$ forwards.

Before we can apply the neutral bits, we first need to compute a partial solution over 16 consecutive steps that can be extended using the neutral bits, which we call *base solution* in our terminology. This base solution is also computed with an offset but it is only one, meaning that it consists of state words $A_{-3}, \ldots, A_{17}$. We can find such a solution using simple message modification, over the message words W_1 to W_{16}, in particular we choose an initial solution for $A_8, \ldots, A_{12}$ which we first extend backwards using words $11, \ldots, 1$ and then forwards using words $12, \ldots, 16$.

Using neutral bits to improve the probabilistic phase. The offset of the base solution being 1, the state may not follow the differential path anymore starting from A_{18}, and the attacker needs to test many different base solutions to go as far as A_{76} and get a complete collision. At first sight, it may therefore seem that we gained only little from this particular use of a start-in-the-middle approach. However, by using a main block offset of 6, there remains freedom that can still be exploited in the message words up to W_{21}. Although their value cannot be changed entirely (as they were fully determined when computing the base solution), we can still use these words to implement neutral bits.

In our attack, we use 51 neutral bits spread on words W_{14} to W_{21}, which are neutral with respect to the computation of state words up to $A_{18\ldots26}$ with good probability. This means that up to the computation of A_{26}, one can take advantage of solutions up to $A_{18\ldots25}$ (that had to be found probabilistically) to find more solutions up to the same step with only a negligible cost. This considerably reduces the complexity of the attack, and we experimentally observed that about 90 % of the computations were done past A_{24}, which can thus be considered to be where the actual probabilistic phase starts. This is a very noticeable improvement from the original A_{17} of the base solution. We give a complete list of these neutral bits in the full version of the paper [19]

There is however one caveat when using neutral bits in such a start-in-the-middle fashion, as one will pay an additional cost in complexity if they interact badly with the base solution when they go through the backward message expansion. In our attack, we chose neutral bits that do so with only a small probability, which can even be lowered to a negligible quantity when running the attack by filtering the base solutions from which to start.

In Figure 4-1, we summarize our application of start-from-the-middle to `SHA-1` with a graphical representation of the attack layout.

4.2   Differential path construction improvements

The full differential path for `SHA-1` collision attacks are made of two parts. The most important part is the linear part built from a combination of local collisions as described by the disturbance vector, which almost covers the last 3 rounds of `SHA-1` and directly contributes a major factor to the overall complexity. The remaining part, the so-called non-linear-part covering mostly round 1, is
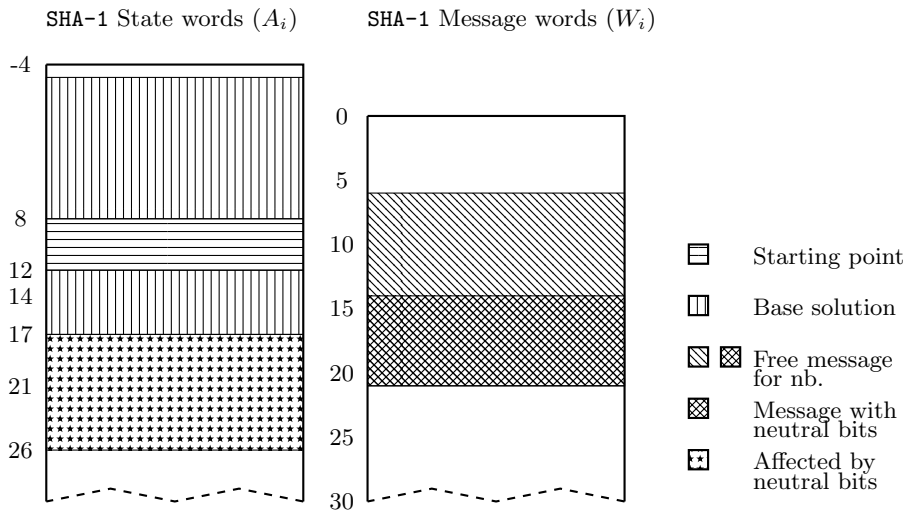
SHA-1 State words (A_i)   **SHA-1** Message words (W_i)

Starting point

Base solution

Free message
for nb.

Message with
neutral bits

Affected by
neutral bits

Fig. 4-1. Illustration of the use of start-from-the-middle for **SHA-1**.

constructed to link up prescribed IV differences and the linear part into a full differential path.

Current methods to construct the non-linear part are a guess-and-determine approach due to De Cannière *et al.* [8] and a meet-in-the-middle approach due to Stevens *et al.* [37]. For the latter an implementation has been made public at Project HashClash [15] that we used for this work. For the linear part, the state-of-the-art is Joint Local-Collision Analysis (JLCA) [39] which analyzes the entire set of differential paths over the last 3 rounds conforming to the disturbance vector and which exploits redundancy to make it practical. Using JLCA one can extract a minimal set of conditions (consisting of starting differences (say for step 20), message bit-relations and ending differences) that leads to the highest probability. Being of the highest probability implies that the factor contribution of the linear part to the overall complexity is minimal, while a minimal set of conditions maximizes the amount of freedom that can be exploited to speed up the attack.

For our attacks we extended (our own implementation of) JLCA to cover *all* steps and to produce the entire set of sufficient state conditions and message bit-relations as used by collision attacks. In particular, we improved JLCA in the following ways:

1. **Include the non-linear part.** Originally JLCA considers the entire set of differential paths that conform to the disturbance vector only over the linear part. This is done by considering sets $\mathcal{Q}_i$ of allowed state differences for each A_i given the disturbance vector (including carries), see [39]. We extended this by defining sets $\mathcal{Q}_i$ for the non-linear part as the state difference given

9

by a previously constructed differential path of the non-linear part. Here one actually has a few options: only consider exact state difference of the non-linear path or also consider changes in carries and/or signs, as well as include state differences conforming to the disturbance vector. We found that allowing changes in carries and/or signs for the state differences given by the non-linear path made JLCA impractical, yet including state differences conforming to the disturbance vector was practical and had a positive effect on the overall probability of the full differential path.

2. **Do not consider auxiliary carries not used in the attack.** Originally JLCA considers local collisions with carries as this improves the overall probability, the probability of variants of paths adding up. However, collision attacks employ sufficient conditions for, say, the first 25 steps, where such auxiliary carries are not used. For these steps JLCA would thus optimize for the wrong model with auxiliary carries. We propose to improve this by not adding up the probability of paths over the first 25 steps, but only to take the maximum probability. We propose to do this by redefining the cumulative probabilities $p_{(\mathcal{P},w)}$ from [39, Sec 4.6] to:

$$p_{(\mathcal{P},w)} = \max_{\widehat{\mathcal{P}}_{[0,25]} \in \mathcal{D}_{[0,25]}} \sum_{\substack{\mathcal{P}' \in \mathcal{D}_{[0,t_e]} \\ \mathcal{P}'|_{[0,25]} = \widehat{\mathcal{P}}_{[0,25]} \\ \mathcal{P} = \mathrm{Reduce}(\mathcal{P}'), w = w(\mathcal{P}')}} \Pr[\mathcal{P}' - \mathcal{P}].$$

In our JLCA implementation this can be simply implemented by replacing the addition of two probabilities by taking their maximum conditional on the current `SHA-1` step.

3. **Determine sufficient conditions.** Originally JLCA only outputted starting differences, ending differences, message bit-relations and the optimal success probability. We propose to extend JLCA to reconstruct the set of differential paths over steps, say, $[0, 25]$, and to determine minimal sets of sufficient conditions and message bit-relations. This can be made possible by keeping the intermediate sets of reduced differential paths $\mathcal{R}_{[t_b,t_e]}$ which were constructed backwards starting at a zero-difference intermediate state of `SHA-1`. Then one can iteratively construct sets $\mathcal{O}_{[0,i)}$ of optimal differential paths over steps $0, \ldots, i-1$, *i.e.*, differential paths compatible with some combination of the optimal starting differences, ending differences and message bit-relations such that the optimal success probability can be achieved. One starts with the set $\mathcal{O}_{[0,0)}$ determined by the optimal starting differences. Given $\mathcal{O}_{[0,i)}$ one can compute $\mathcal{O}_{[0,i+1)}$ by considering all possible extensions of every differential path in $\mathcal{O}_{[0,i)}$ with step i (under the predefined constraints, i.e. $\Delta Q_j \in \mathcal{Q}_j$, $\delta W_i \in \mathcal{W} - i$, see [39]). From all those paths, one only stores in $\mathcal{O}_{[0,i+1)}$ those that can be complemented by a reduced differential path over steps $i + 1, \ldots, t_e$ from $\mathcal{R}_{[i+1,t_e]}$ such that the optimal success probability is achieved over steps $0, \ldots, t_e$.

Now given, say, $\mathcal{O}_{[0,26)}$, we can select any path and determine its conditions necessary and sufficient for steps $0, \ldots, 25$ and the optimal set of message bit-relations that goes with it. Although we use only one path, having the

entire set $\mathcal{O}_{[0,26)}$ opens even more avenues for future work. For instance, one might consider an entire subclass of 2^k differential paths from $\mathcal{O}_{[0,26)}$ that can be described by state conditions linear in message bits and a set of (linear) message bit-relations. This would provide k bits more in degrees of freedom that can be exploited by speed up techniques.

In short, we propose several extensions to JLCA that allows us to determine sufficient state conditions and message bit-relations optimized for collision attacks, *i.e.* minimal set of conditions attaining the highest success probability paths (where auxiliary carries are only allowed after a certain step).

4.3 Implementation of the attack on GPUs

We now present a high-level view of the implementation of our attack, focusing on the features that make it efficient on GPUs. Their architecture being noticeably different from the one of CPUs, we first recall a few important points that will help understanding the design decisions[4].

Number of cores and scheduling. A modern GPU can feature more than a thousand of small cores, that are packed together in a small number of larger "multiprocessor" execution units. Taking the example of the Nvidia GTX 970 for concreteness, there are 13 multiprocessors of 128 cores each, making 1664 cores in total [33]. The fastest instructions (such as for instance 32-bit bitwise logical operations or modular addition) have a throughput of 1 per core, which means that in ideal conditions 1664 instructions may be simultaneously processed by such a GPU in one clock cycle [32].

Yet, so many instructions cannot be emitted independently, or to put it in another way, one cannot run an independent thread of computation for every core. In fact, threads are grouped together by 32 forming a *warp*, and only warps may be scheduled independently. Threads within a warp may have a diverging control flow, for instance by taking a different path upon encountering a conditional statement, but their execution in this case is serialized. At an even higher level, warps executing the same code can be grouped together as *blocks*.

Furthermore, on each multiprocessor one can run up to 2048 threads simultaneously, which are dynamically scheduled every cycle onto the 128 cores at a warp granularity. Thus while a warp is waiting for the results of a computation or for a (high latency) memory operation to return, another warp can be scheduled. Although having more threads does not increase the computational power of the multiprocessor, such overbooking of cores can be used to hide latencies and thus increase efficiency of a GPU program.

In short, to achieve an optimal performance, one must bundle computations by groups of 32 threads executing the same instructions most of the time and diverging as little as possible and use as many threads as possible.

---

[4] We specifically discuss these points for Nvidia GPUs of the most recent *Maxwell* generation such as the GTX 970 being used in our attacks.

Memory architecture and thread synchronization. In the same way as they feature many execution units, GPUs also provide memory of a generous size, which must however be shared among the threads. The amount of memory available to a single thread is therefore much less than what is typically available on a CPU (it of course highly depends on the number of running threads, but can be lower than 1 MB). This, together with the facts that threads of a same warp do not actually execute independently of each other and that threads of a same block run the same code makes it enticing to organize the memory structure of a program at the block level. Fortunately, this is made rather easy by the fact that many efficient synchronization functions are available for the threads, both at the warp and at the block level.

Balancing the work between the GPU and the CPU. The implementation of our attack can be broadly decomposed in two phases. The first step consists in computing a certain number of base solutions as in Section 4.1 and in storing them on disk. Because the total number of base solutions necessary to find a collision in our attack is rather small (about 2^{25}) and because they can be computed quickly, this can be done efficiently in an offline fashion using CPUs.

The second phase then consists in trying to extend probabilistically the base solutions (and their variants through the use of neutral bits) to find a collision. This is an intensely parallel task that is well suited to run on GPUs. However, as it was emphasized above, GPUs are most efficient when there is a high coherency between the execution of many threads. For that reason, we must avoid having idle threads that are waiting because their candidate solutions failed to follow the differential paths, while others keep on verifying a more successful one. Our approach to this is to fragment the verification into many small pieces (or *snippets*) that are chosen in a way which ensures that coherency is maintained for every thread of a warp when executing a single snippet, except in only a few small points. This is achieved through a series of intermediary buffers that store inputs and outputs for the snippets; a warp then only executes a given snippet if enough inputs are available for every of its threads. One should note that there is no need to entirely decompose the second step of the attack into snippets, and that a final part can again be run in a more serial fashion. Indeed, if inputs to such a part are scarce, there is no real advantage in verifying them in a highly parallel way.

The sort of decomposition used for the GPU phase of our attack as described above is in no way constrained by the specifics of our attack. In fact, it is quite general, and we believe that it can be successfully applied to many an implementation of cryptographic attacks. We conclude this section by giving more details of the application of this approach to the case of SHA-1.

Choice of the snippets. As it was mentioned in Section 4.1, our attack uses neutral bits acting on the state words of step 18 to 26. The choice we made for the decomposition into snippets reflects this use of neutral bits: we use intermediary

buffers to store partial solutions up to step 17, 18, etc. Then for each step the corresponding snippet consists in loading one partial solution per thread of a warp and applying every combination of neutral bits for this step. Each combination is tried by every thread at the same time on its own partial solution, thereby maintaining coherency. Then, each thread writes every resulting partial solution extended by one step to the output buffer of the snippet (which is the input buffer of the next snippet) at the condition that it is indeed valid, this being the only part of the code where threads may briefly diverge. For the later steps when no neutral bits can be used anymore, the snippets regroup the computation of several steps together, and eventually the verification that partial solutions up to step 56 make valid collisions is done on a CPU. This is partly because the amount of available memory makes it hard to use step-by-step snippets until the end, but also because such partial solutions are only produced very slowly (a single GTX 970 produces solutions up to step 56 at a speed of about 0.017 solution per second, that is about 1 per minute).

Complete process of the attack. When running the attack, every warp tries to work with partial solutions that are up to the latest step possible. If no work is available there, it tries to work with partial solutions up to the second-latest step, etc. Eventually warps resort to using base solutions in the worst case that no work is available anywhere else. As was already said in Section 4.1, we experimentally observed that most of the work is done on partial solutions that are at least up to step 24, and work on solutions up to lower steps (and in particular base solutions) is thus done only intermittently. We conclude this description by giving a simplified flow chart (made slightly incorrect for the sake of clarity) of the GPU part of the SHA-1 attack in Figure 4-2.
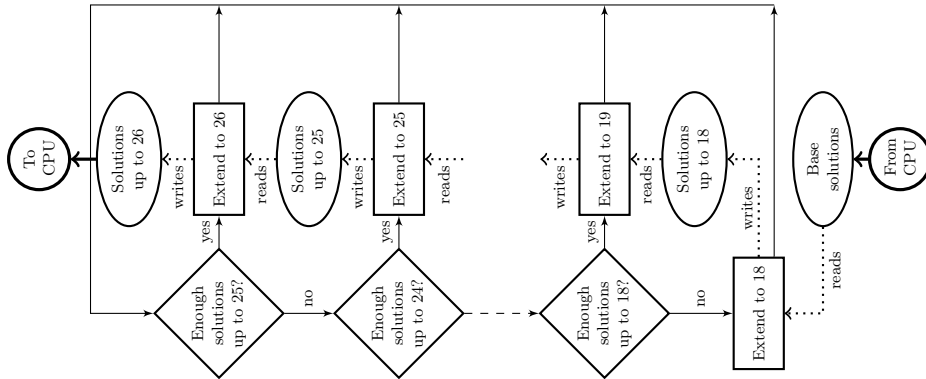
Fig. 4-2. Simplified flow chart for the GPU part of the attack. The start of this infinite loop is in the top left corner. Rectangles "□" represent snippets, ellipses "◯" represent shared buffers, plain lines "↗" represent control flow, and dotted lines "⌐" represent data flow.

5 Details of the attack and its implementation

5.1 The case of `SHA-1`

For our 76-step free-start collision attack, we selected disturbance vector II(55,0) (following Manuel's classification [23]), and this for two reasons. Firstly, JLCA showed it to be one of the best for 76-steps. Secondly, the required IV difference is very sparse and localized on the two lowest bit positions, thus having low potential for interference of the neutral bits with state conditions on the first few steps.

As explained in Section 4.2, we have extended JLCA to determine optimal sets of state conditions and message bit-relations given a non-linear path. For our attack we tried both non-linear differential path construction methods, *i.e.* the guess-and-determine method using our own implementation, and the meet-in-the-middle method using the public HashClash implementation [15]. We have found that the meet-in-the-middle approach generally resulted in fewer conditions and that furthermore we could better position the conditions. Our initial non-linear path was thus generated using the meet-in-the-middle approach, although when considering the full differential path one can encounter contradictions in the message bit-relations and/or an unsolvable highest density part of the differential path. These are expected situations which are easily solvable by considering variations of the non-linear part, which we did using the guess-and-determine approach.

The sufficient conditions over steps 0–35 and the message bit-relations can be found in the full version [19].

5.2 GPU implementation

We complete the description of our approach towards GPU programming from Section 4.3 with a few lower-level details about our implementation on GTX 970.

Block layout. A GTX 970 features 13 multiprocessors of 128 cores. Each multiprocessor can host a maximum of 2048 threads regrouped in at least 2 and at most 32 blocks [32]. If every multiprocessor of the GPU hosts 2048 threads, we say that we have reached *full occupancy*. While a multiprocessor can only physically run one thread per core (*i.e.* 128) at a given time, a higher number of resident threads is beneficial to hide computation and memory latencies. These can have a significant impact on the performance as a single waiting thread causes its entire warp of 32 to wait with him; it is thus important in this case for the multiprocessor to be able to schedule another warp in the meantime.

Achieving full occupancy is not however an absolute objective as it may or may not result in optimal performance depending on the resources needed by every thread. Important factors in that respect are the average amount of memory and the number of registers needed by a single thread, both being resources shared among the threads. In our implementation, the threads need to run rather heavy functions and full occupancy is typically not desirable. One

reason why it is so is that we need to allocate 64 registers per thread in order to prevent register spilling in some of the most expensive functions; a multiprocessor having "only" 2^{16} registers, this limits the number of threads to 1024. As a result, we use a layout of 26 blocks of 512 threads each, every multiprocessor being then able to host 2 such blocks.

Buffer framework. As it was already said in Section 4.3, we use a number of shared buffers in our implementation in order to maximize coherency among threads of a single warp. With the exception of the buffers holding the base solutions and the collision candidates, there is one instance of every buffer per block. This allows to use block-wise instead of global synchronization mechanisms when updating the buffers' content, thence reducing the overhead inherent to the use of such shared data structures.

All of the buffers are cyclic and hold 2^{20} elements of a few different type and size (with the exception of the ones holding solutions after step A_{36} which are of size only 2^{10}, given their limited number). The different types of buffers are the following:

– The base solution buffer contains the value of 6 words of the solution's state A_{12} to A_{17}, and the 16 message words W_6 to W_{21}, making 22 words in total. Although only 5 state words are necessary to fully determine the base solution, the value of A_{12} is additionally needed for the computation of some of the neutral bits.
– An *extended base solution* buffer is used after the step A_{21}; it holds the value of state words A_{17} to A_{21}, message words W_{14} to W_{18} and W_{20}, and the index of the base solution that it extends, using 11 words in total.
– For all the remaining steps with neutral bits, a compact representation is used that only refers to the (extended) base solution from which it is derived and the value of its active neutral bits; all of this can be stored in only two words.
– A candidate solution buffer of 5 state words and 16 message words is used for partial solutions up to step A_{36} and step A_{56}.

The decomposition into base and extended base solutions was carefully chosen from the position of the neutral bits. From their description [19], one can see that neutral bits on the message words up to W_{18} are only used up to step A_{21}; similarly, neutral bits on the words W_{19} to W_{21} are only used after step A_{21}. It is then only natural to define extended base solutions as up to A_{21}. Of course one could have dispensed with such a decomposition altogether, but this would mean that extending a base solution to the later steps (say A_{24}) would systematically need to start recomputing many of the earlier steps from A_{17} before being able to do any useful work and this would be an unnecessary burden on these critical steps. We describe our packing of the neutral bits and of the index to the (extended) base solution in the full version [19]. As a side-note, let us also mention that the use of A_{36} and A_{56} as boundaries for the candidate solutions simply comes from the fact that each is the last of a series of 5 state words with no differences.

On the pure implementation side, we also carefully took into account the presence of a limited amount of very fast multiprocessor-specific shared memory. While the 96 KB available per multiprocessor is hardly enough to store the whole buffers themselves, we take advantage of it by dissociating the storage of the buffers and of the meta-data used for their control logic, the latter being held in shared memory. This improves the overall latency of buffer manipulations, especially in case of heavy contention between different warps. This local shared memory is also very useful to buffer the writes to the buffers themselves. Indeed, only a fraction (often as low as $\frac{1}{8}$) of the threads of a warp have a valid solution to write after having tested a single candidate, and the more unsuccessful threads need to wait while the former write their solution to global memory. It is therefore beneficial to first write the solutions to a small local warp-specific buffer and to flush it to the main block-wise buffer as soon as it holds 32 solutions or more, thence significantly reducing the number of accesses to the slower global memory.

GPU tuning. After our initial implementation, we did some fine tuning of the GPU BIOS settings in order to try having an optimal performance. One first objective was to ensure that the GPU fans work at 100% during the attack, as this was strangely not the case initially, and was obviously not ideal for cooling. We also experimented with various temperature limits (that define when the GPU will start to throttle) and both over-clocking and under-volting. Taken together, these variations can have a significant impact on the overall performance of the program, as can be seen with our 76-step attack below.

6 Results and perspectives

In this last section, we give the statistics for the performance of our implementation of the 76-step attack and estimate the cost of a collision on the full compression function of `SHA-1` using similar methods.

6.1 The 76-step collisions

The first collision was found when running the attack on a single GPU. Based on the production rate of partial solutions up to step 56, the estimated time to find a collision was slightly less than 5 days, at 4.94 days. This rate was also observed in practice, although we also witnessed significant outliers; as a matter of fact, the first collision was found in less than two days.

We subsequently ran the attack for a longer time on a server with four GPUs, and found 17 additional collisions. By improving the implementation and the GPU settings, we managed to significantly decrease the average time needed to find a collision. For the best configuration we found, the best-performing GPU computed collisions at an expected rate of 1 every 4.16 days, with an average of 4.42 for the 4 GPUs (producing solutions up to step 56 at a rate of 0.0171 per second). The whole server could then be expected to produce one collision every 1.1 day. Our GPU implementation of `SHA-1` can compute about $2^{31.8}$ `SHA-1`

compression functions per second. This means that on the best-performing GPU our attack has a complexity equivalent to $2^{50.25}$ calls to the compression function. If one takes the average over the 4 GPUs, this increases slightly to $2^{50.34}$.

We also implemented our attack to run on a standard CPU, which provides an interesting comparison of the relative performance of the attack versus the speed of raw `SHA-1` computations. On an Haswell Core-i5 running at $3.2\,\text{GHz}$, the OpenSSL implementation of `SHA-1` can compute $2^{23.47}$ compression functions per second, while our attack program generates solutions up to step 56 at a rate of 0.000124 per second. The total complexity of the attack thus requires about 606.12 core-days and has a complexity of $2^{49.1}$ compression function calls. This means that a single GTX 970 is worth 322 such CPU cores when computing the `SHA-1` compression function, and 138 cores when running our attack program (this increases to 146 for our best-performing GPU). While this drop in relative efficiency was to be expected, it is somehow surprisingly small given the complexity of our implementation and *e.g.* the intensive use of large shared data structures. Our careful implementation thus gives a much better value for the GPUs when compared to previous attempts at running cryptographic attacks on such a platform; in their attack, Grechnikov and Adinetz estimated a GPU to be worth 39 CPU cores [14].

6.2   Collisions on the full compression function

We are currently working to apply our methods to a free-start collision attack for the full `SHA-1`. Precisely estimating the cost of such an attack is always difficult before it is actually implemented as several factors may influence the complexity; none the least is the number and the quality of the neutral bits (or of accelerating techniques in general), which is typically hard to determine without a full implementation. We can however provide rather reliable estimates for different disturbance vectors by comparing the cost of the linear parts, as well as the number of conditions over the non-linear parts, and by making an educated guess of where should be the last step with a significant number of neutral bits. This guess is in particular made easier by the fact that we can compare a candidate disturbance vector to the one used for the 76-step attack, for which we have very precise results. As a consequence, we get the estimates in Table 6-1 for the complexity of an attack starting at A_{25} for two disturbance vectors. These figures need to be modulated by the fact that different DVs may yield neutral bits of different quality. Both II(55,0) and I(51,0) result in IVs with two differences, though the ones of II(55,0) may be at better positions. As a consequence, one may need to include step A_{24} and its one condition in the critical computations for I(51,0), thus doubling the complexity. Things are even worse for II(51,0) which yields an IV with five differences. Consequently, one would expect neutral bits to be markedly less efficient, and should probably add the cost of both A_{24} and A_{23}, resulting in a 6-bit increase of complexity. Thus, based on these two DVs, we can expect to find a free-start collision for 80 steps for an approximate cost of 7218 GPU-days based on I(51,0), and 8234 GPU-days using II(51,0). With a cluster of 64 GPUs, this represents 4 months

17

Table 6-1. Complexity comparison and estimates for an 80-step attack. #C denotes the number of conditions for a given step and Gd is short for GPU-day (the cost as a number of compression function computation is also given as an alternative measure). The use of † denotes an estimated cost.

DV	Steps	Prob. (A_{25})	Cost (A_{25})	#C (A_{24})	#C (A_{23})
II(55,0)	76	$2^{-52.59}$	4.4 Gd ($2^{50.3}$)	1	3
I(51,0)	80	$2^{-62.27}$	$3609^{\dagger}$ Gd ($2^{60\,\dagger}$)	1	2
II(51,0)	80	$2^{-57.46}$	$129^{\dagger}$ Gd ($2^{55.2\,\dagger}$)	3	3

of computation or thereabouts. While this gives us a reasonable upper-bound, it is still rather high and hence not entirely satisfactory. We plan to significantly improve the complexity of such an attack by:

1. investigating better disturbance vectors such as II(56,0), II(58,0) or II(59,0); unfortunately computing their exact probability with JLCA is much harder than for *e.g.* II(51,0);
2. using better accelerating techniques than the rather simple neutral bits used so far for the 76-step attack.

Both options should result in quite better attacks than the estimates from above. This is a promising and exciting future work, and we hope to achieve significant results in the near future.

References

1. Biham, E., Chen, R.: Near-Collisions of SHA-0. In: Franklin, M.K. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 3152, pp. 290–305. Springer (2004)
2. Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., Jalby, W.: Collisions of SHA-0 and Reduced SHA-1. In: Cramer [5], pp. 36–57
3. Brassard, G. (ed.): CRYPTO, Lecture Notes in Computer Science, vol. 435. Springer (1990)
4. Chabaud, F., Joux, A.: Differential Collisions in SHA-0. In: Krawczyk, H. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 1462, pp. 56–71. Springer (1998)
5. Cramer, R. (ed.): EUROCRYPT, Lecture Notes in Computer Science, vol. 3494. Springer (2005)
6. Damgård, I.: A Design Principle for Hash Functions. In: Brassard [3], pp. 416–427
7. De Cannière, C., Mendel, F., Rechberger, C.: Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In: Adams, C.M., Miri, A., Wiener, M.J. (eds.) SAC. Lecture Notes in Computer Science, vol. 4876, pp. 56–73. Springer (2007)
8. De Cannière, C., Rechberger, C.: Finding SHA-1 Characteristics: General Results and Applications. In: Lai, X., Chen, K. (eds.) ASIACRYPT. Lecture Notes in Computer Science, vol. 4284, pp. 1–20. Springer (2006)

9. den Boer, B., Bosselaers, A.: An Attack on the Last Two Rounds of MD4. In: Feigenbaum, J. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 576, pp. 194–203. Springer (1991)

10. den Boer, B., Bosselaers, A.: Collisions for the Compression Function of MD5. In: Helleseth, T. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 765, pp. 293–304. Springer (1993)

11. Dobbertin, H.: Cryptanalysis of MD4. In: Gollmann, D. (ed.) FSE. Lecture Notes in Computer Science, vol. 1039, pp. 53–69. Springer (1996)

12. Gilbert, H., Peyrin, T.: Super-Sbox Cryptanalysis: Improved Attacks for AES-Like Permutations. In: Hong, S., Iwata, T. (eds.) FSE. Lecture Notes in Computer Science, vol. 6147, pp. 365–383. Springer (2010), http://dx.doi.org/10.1007/978-3-642-13858-4

13. Grechnikov, E.A.: Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics. IACR Cryptology ePrint Archive 2010, 413 (2010)

14. Grechnikov, E.A., Adinetz, A.V.: Collision for 75-step SHA-1: Intensive Parallelization with GPU. IACR Cryptology ePrint Archive 2011, 641 (2011)

15. Hashclash project webpage, https://marc-stevens.nl/p/hashclash/

16. Jean, J., Naya-Plasencia, M., Peyrin, T.: Improved Rebound Attack on the Finalist Grøstl. In: Canteaut, A. (ed.) FSE. Lecture Notes in Computer Science, vol. 7549, pp. 110–126. Springer (2012), http://dx.doi.org/10.1007/978-3-642-34047-5

17. Johansson, T., Nguyen, P.Q. (eds.): EUROCRYPT, Lecture Notes in Computer Science, vol. 7881. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38348-9

18. Joux, A., Peyrin, T.: Hash Functions and the (Amplified) Boomerang Attack. In: Menezes, A. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 4622, pp. 244–263. Springer (2007)

19. Karpman, P., Peyrin, T., Stevens, M.: Practical Free-Start Collision Attacks on 76-step SHA-1. IACR Cryptology ePrint Archive 2015, 530 (2015)

20. Klíma, V.: Tunnels in Hash Functions: MD5 Collisions Within a Minute. IACR Cryptology ePrint Archive 2006, 105 (2006)

21. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In: Matsui, M. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 5912, pp. 126–143. Springer (2009), http://dx.doi.org/10.1007/978-3-642-10366-7

22. Landelle, F., Peyrin, T.: Cryptanalysis of Full RIPEMD-128. In: Johansson and Nguyen [17], pp. 228–244, http://dx.doi.org/10.1007/978-3-642-38348-9

23. Manuel, S.: Classification and generation of disturbance vectors for collision attacks against SHA-1. Des. Codes Cryptography 59(1-3), 247–263 (2011)

24. Mendel, F., Peyrin, T., Rechberger, C., Schläffer, M.: Improved Cryptanalysis of the Reduced Grøstl Compression Function, ECHO Permutation and AES Block Cipher. In: Jr., M.J.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC. Lecture Notes in Computer Science, vol. 5867, pp. 16–35. Springer (2009), http://dx.doi.org/10.1007/978-3-642-05445-7

25. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In: Dunkelman, O. (ed.) FSE. Lecture Notes in Computer Science, vol. 5665, pp. 260–276. Springer (2009), http://dx.doi.org/10.1007/978-3-642-03317-9

26. Mendel, F., Rijmen, V., Schläffer, M.: Collision Attack on 5 Rounds of Grøstl. In: Cid, C., Rechberger, C. (eds.) FSE. Lecture Notes in Computer Science, vol. 8540, pp. 509–521. Springer (2014), http://dx.doi.org/10.1007/978-3-662-46706-0

27. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard [3], pp. 428–446
28. National Institute of Standards and Technology: FIPS 180: Secure Hash Standard (May 1993)
29. National Institute of Standards and Technology: FIPS 180-1: Secure Hash Standard (April 1995)
30. National Institute of Standards and Technology: FIPS 180-2: Secure Hash Standard (August 2002)
31. National Institute of Standards and Technology: Draft FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (May 2014)
32. Nvidia Corporation: Cuda C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide
33. Nvidia Corporation: Nvidia Geforce GTX 970 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications
34. Rivest, R.L.: The MD4 message digest algorithm. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO. Lecture Notes in Computer Science, vol. 537, pp. 303–311. Springer (1990)
35. Rivest, R.L.: RFC 1321: The MD5 Message-Digest Algorithm (April 1992)
36. Saarinen, M.O.: Cryptanalysis of Block Ciphers Based on SHA-1 and MD5. In: Johansson, T. (ed.) FSE. Lecture Notes in Computer Science, vol. 2887, pp. 36–44. Springer (2003)
37. Stevens, M.: Attacks on Hash Functions and Applications. Ph.D. thesis, Leiden University (June 2012)
38. Stevens, M.: Counter-Cryptanalysis. In: Canetti, R., Garay, J.A. (eds.) CRYPTO. Lecture Notes in Computer Science, vol. 8042, pp. 129–146. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40041-4
39. Stevens, M.: New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis. In: Johansson and Nguyen [17], pp. 245–261, http://dx.doi.org/10.1007/978-3-642-38348-9
40. Stevens, M., Lenstra, A.K., de Weger, B.: Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In: Naor, M. (ed.) EURO-CRYPT. Lecture Notes in Computer Science, vol. 4515, pp. 1–22. Springer (2007), http://dx.doi.org/10.1007/978-3-540-72540-4_1
41. Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A.K., Molnar, D., Osvik, D.A., de Weger, B.: Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In: Halevi, S. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 5677, pp. 55–69. Springer (2009), http://dx.doi.org/10.1007/978-3-642-03356-8
42. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 3621, pp. 17–36. Springer (2005)
43. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer [5], pp. 19–35