# Speeding up detection of SHA-1 collision attacks using unavoidable attack conditions

Marc Stevens, *CWI;* Daniel Shumow, *Microsoft Research*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stevens

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

# Speeding up detection of SHA-1 collision attacks using unavoidable attack conditions

Marc Stevens
*CWI Amsterdam*

Daniel Shumow
*Microsoft Research*

## Abstract

Counter-cryptanalysis, the concept of using cryptanalytic techniques to detect cryptanalytic attacks, was introduced at CRYPTO 2013 [23] with a hash collision detection algorithm. That is, an algorithm that detects whether a given *single* message is part of a colliding message pair constructed using a cryptanalytic collision attack on MD5 or SHA-1.

Unfortunately, the original collision detection algorithm is not a low-cost solution as it costs 15 to 224 times more than a single hash computation. In this paper we present a significant performance improvement for collision detection based on the new concept of *unavoidable conditions*. Unavoidable conditions are conditions that are *necessary for all feasible attacks* in a certain attack class. As such they can be used to quickly dismiss particular attack classes that may have been used in the construction of the message. To determine an unavoidable condition one must rule out any feasible variant attack where this condition might not be necessary, otherwise adversaries aware of counter-cryptanalysis could easily bypass this improved collision detection with a carefully chosen variant attack. Based on a conjecture solidly supported by the current state of the art, we show how we can determine such unavoidable conditions for SHA-1.

We have implemented the improved SHA-1 collision detection using such unavoidable conditions and which is more than 20 times faster than without our unavoidable condition improvements. We have measured that overall our implemented SHA-1 with collision detection is only a factor 1.60 slower, on average, than SHA-1. With the demonstration of a SHA-1 collision, the algorithm presented here has been deployed by Git, GitHub, Google Drive, Gmail, Microsoft OneDrive and others, showing the effectiveness of this technique.

## 1 Introduction

Cryptographic hash functions, computing a small fixed-size hash value for a given message of arbitrary length, are a crucial cryptographic primitive that are used to secure countless systems and applications. A key cryptographic requirement is that it should be computationally infeasible to find *collisions*: two distinct messages with the same hash value. Industry's previous de facto choices MD5 and SHA-1 are both based on the Merkle-Damgård construction [18, 6] that iterates a compression function that updates a fixed-size internal state called the chaining value (CV) with fixed-size pieces of the input message.

In 2004, MD5 was completely broken and real collisions were presented by Wang et al.[33, 35]. Their collision attack consisted of two so-called *near-collision attacks* on MD5's compression function where the first introduces a difference in the chaining value and the second eliminates this difference again. Hence, these so-called *identical-prefix collisions* had a limitation that the two colliding messages need to be identical before and after these near-collision blocks. In 2007 Stevens et al.[27] introduced *chosen-prefix collisions* for MD5 that allowed arbitrary different prefixes. Irrefutable proof that hash function collisions indeed form a realistic and significant threat to Internet security was presented at CRYPTO 2009 by Stevens et al. [29] by demonstrating a certificate authority that could issue two Certs with different keys that have the same hash value.

More proof of the threat posed by collision attacks appeared in 2012 when it became clear that not only academic efforts have been spent on breaking hash functions. Nation-state actors [20, 14, 13] have been linked to the highly advanced espionage malware, so named Flame, that was found targeting the Middle-East in May 2012. As it turned out, it used a forged signature to craft malicious windows updates.

Despite the common knowledge that MD5 is insecure for digital signatures effectively since 2004, even in 2017 there are still Industry issues in deprecating MD5 for signatures [10]. SHA-1, designed by NSA and standardized by NIST [19], is also weak and was theoretically broken in 2005 with a collision attack with an estimated complexity of $2^{69}$ SHA-1 calls presented by Wang et al.[34]. With real collisions for full SHA-1 out of reach at that time, there were efforts at producing collisions for reduced versions of SHA-1: 64 steps [3] (with a cost of $2^{35}$ SHA-1 calls), 70 steps [2] (cost $2^{44}$ SHA-1), 73 steps [8] (cost $2^{50.7}$ SHA-1), the last being 75 steps [9] (cost $2^{57.7}$ SHA-1) in 2011. The cost of collisions for SHA-1 was improved to $2^{61}$ SHA-1 calls at EUROCRYPT 2013 [24], together with a near-collision attack with cost $2^{57.5}$ and a chosen-prefix collision attack with cost $2^{77.1}$ SHA-1 calls, which remains the current state-of-the-art. Other recent efforts focused on finding *freestart collisions* for SHA-1, i.e., collisions for its compression function, with a 76-step freestart collision [12] (cost $2^{50}$ SHA-1) and more recently a freestart collision for full SHA-1 [26]. Despite various efforts, an actual collision for SHA-1 remained out of reach for 11 years, but this year a SHA-1 collision was finally announced by Stevens et al.[25]. This shows that SHA-1 collision attacks have finally become practical. Furthermore, they provided several examples of PDF file pairs that have the same SHA-1 hash, yet show distinct visual contents.

At CRYPTO 2013 the novel concept *counter-cryptanalysis* of using cryptanalytic techniques to detect cryptanalytic attacks was introduced in the form of a hash collision detection algorithm [23]. This hash collision detection algorithm is able to expose cryptanalytic collision attacks given only one message from a colliding message pair. It's utility was immediately proven by using it to uncover the *yet unknown* chosen-prefix collision attack in Flame's forged signature, regardless of the fact that its sibling colliding file remains unknown. Moreover, it even enabled forensic analysis by recovering the internal differential paths, which were used in a reconstruction of the attack procedure and complexity [7].

In principle the collision attack detection provides strong guarantees: it guarantees detection of any variant collision attack in each tested attack class, whereas the chance of false positives is as negligible as the chance of finding a random second preimage. However, so far there is a significant cost: to detect collision attacks against SHA-1 (respectively MD5) costs the equivalent of hashing the message 15 (respectively 224) times, detecting the 15 (respectively 224) best attack classes.

## 1.1 Motivation

The main motivation of this paper is to provide an effective manner to reduce the potential harm of SHA-1 collision attacks for the near future as discussed below. It is not the aim of SHA-1 collision detection to obviate the need to move to newer hash functions with longer digests. Rather, SHA-1 collision detection is meant to be a mitigation used for deployed systems that are unable to migrate to a new hash function. In these cases, an implementation of SHA-1 with collision detection may be used as a drop in replacement. Such an update, that requires only changing the module responsible for hashing, is significantly easier than redeploying an entire distributed system, including revising protocols that currently rely on SHA-1. Collision detection for SHA-1 attacks is a thorough stop-gap solution that will provide security to systems and software that may not be able to migrate to newer hash functions before SHA-1 collisions become a viable security threat. The example of the well-known version control system Git that relies very strongly on SHA-1 for integrity and even security is very amenable to such a solution. In fact at the time of writing, Git and GitHub now use the improved SHA-1 collision detection of this paper by default. Our improved implementation is also being used by Google Drive, Gmail and Microsoft OneDrive.

Based on the latest results for the complexity of finding a SHA-1 collision, the projected cost of such an attack ranges from US\$ 75 K and US\$ 120 K by renting low-cost computing resources on Amazon EC2 [22, 26], which is significantly lower than Schneier's 2012 estimates. These projections resulted in the withdrawal of CABForum Ballot 152 to extend issuance of SHA-1 based HTTPS certificates, and in the deprecation of SHA-1 for digital signatures in the IETF's TLS protocol specification version 1.3. The recent SHA-1 collision paper further confirms these costs [25].

Unfortunately CABForum restrictions on the use of SHA-1 do not apply on Certification Authority certificates currently outside their CA program. E.g., it excludes retracted CA certificates that are still supported by older systems (and CA certificates have indeed been retracted to circumvent CABForum regulations and continue to issue new SHA-1 certificates [1] to serve to these older systems), and certificates for other TLS applications including up to 10% of credit card payment systems [31]. It thus remains in widespread use across the software industry for, e.g., digital signatures on software, documents, and many other applications, perhaps most notably in the widely used Git versioning system.

It is very likely that SHA-1 is heading towards a similar fate as MD5, risking various security issues for many years to come. Certainly, the spectacular end of life of MD5, including a high profile cyberattack on the nation state level, provided advanced warning of the end of life of SHA-1. Indeed, the success of cryptanalytic attacks of

[1]E.g., SHA-1 certificates are still being sold by CloudFlare at the time of writing: www.cloudflare.com/ssl/dedicated-certificates/.

the Merkle-Damgard construction motivated the SHA-3 competition. Not to mention, inspired widespread efforts to migrate deployed software to the longer length digest hash functions of SHA-2 family. So, one may challenge the utility of collision detection for the SHA-1 function, which has been known to have an impending break for some time. However, the Flame attacks show that long after newer versions of software have been deployed, older versions that rely on older cryptography may still be in use and provide a vulnerability for attackers. Even with this cautionary tale, as noted above, various software and services still issue SHA-1 certificates or use SHA-1. So even though SHA-1 collisions have been expected for some time, it has not been sufficient to motivate a complete migration to newer hash functions. Even if systems are moved from using SHA-1, verification of signatures of SHA-1 digests may remain necessary for existing signatures, such as deployed binaries or not yet expired certificates.

An example is GPG/PGP email and attachment signatures where SHA-1-based signatures remain common. E.g., Stevens et al.'s colliding PDF document technique would allow an attacker to have someone sign and email a carefully crafted benign PDF document with GPG/PGP using a SHA-1-based signature. That signature would then also be valid for a malicious PDF document that was crafted together with the benign PDF document to make them collide. Also, as previously mentioned, another example of widely a deployed system that relies on SHA-1 in a fundamental way is Git, which uses SHA-1 as an identifier for commits. It is infeasible that all deployed Git repositories will be migrated off of SHA-1, but since SHA-1 collisions are now feasible Git might be at risk. As one potentially scenario, consider an attacker that has committed one file of a colliding pair[2] to a Git repository under his control, in which case he could then selectively deliver either contents to targeted users, without the users noticing by looking at Git hashes or verifying signatures on the repository. Although, Git now uses this SHA-1 collision detection algorithm, so this risk has been mitigated for updated clients.

## 1.2 Collision detection

Collision attack detection exploits two key facts-of-life for feasible cryptanalytic collision attacks on MD5 and SHA-1. The first is a requirement for a high-probability differential path, which necessarily includes a section with no differences (or MSB-differences for MD5) to achieve the high-probability. The second is the direct consequence that there only few message block differences that admit such a high-probability differential path. Extensive studies for message block differences that allow high-probability differential paths for both MD5 and SHA-1 strongly confirm these properties.

Collision detection detects near-collision attacks against MD5's or SHA-1's compression function for a given message by 'jumping' from the current compression function evaluation $CV_{out} = Compress(CV_{in}, B)$ to a presumed related compression function evaluation $CV'_{out} = Compress(CV'_{in}, B')$. If $B$ (and $B'$) were constructed using a collision attack that uses message block difference $\delta B$ and trivial difference $\delta WS_i$ at step $i$ then the presumed related compression function evaluation can be fully reconstructed. Namely, those differences directly imply values for message block $B' = B + \delta B$ and state $WS'_i = WS_i + \delta WS_i$ at step $i$, which are sufficient to compute the related input chaining value $CV'_{in}$ and thereby also the related output chaining value $CV'_{out}$. This reconstruction from the middle of the related compression function evaluation is called a *recompression*. A collision attack necessarily requires a final near-collision attack with $CV'_{out} = CV_{out}$, which can be detected in this manner. If no collision attack was used to construct $B$ then finding $CV'_{out} = CV_{out}$ means that we have found a second pre-image for the compression function by chance. Therefore the chance of false positives is as negligible as the chance of finding a random second preimage.

For MD5 and SHA-1 one thus distinguishes many attack classes that each are described by the message block difference $\delta B$, step $i$ and intermediate state difference $\delta WS_i$. In the case of SHA-1 each attack class depends entirely on the so-called *disturbance vector* (DV). In either case, for every block of the given message, each attack class requires another compression function evaluation. With the 223 known attack classes for MD5, collision detection costs a factor 224 more than MD5. SHA-1 collision detection costs a factor 15 more than SHA-1 given the original proposed list of 14 most threatening disturbance vectors.

## 2 Our contributions

In this paper we present a significant run-time performance improvement to collision detection. This improvement is based on a new concept in cryptanalysis, namely *unavoidable conditions*, which are conditions that are necessary for *all* feasible attacks within a certain class of attacks. To determine an unavoidable condition one must rule out any feasible variant attack where this condition might not be necessary. Otherwise, adversaries aware of counter-cryptanalysis could easily bypass this improved collision detection with a carefully chosen variant attack.

[2] As Git adds a header before computing the SHA-1 hash of an object, this header should be taken into account while constructing the SHA-1 collision.

We provide a formal framework of unavoidable conditions for collision attacks on MD5-like compression functions that can be used to show that indeed conditions are unavoidable, and we show how they can be used to speed up collision detection.

Furthermore, we present a conjecture that SHA-1 collision attacks based on a disturbance vector may not deviate from the prescribed local collisions for steps 35 up to 65 to remain feasible. As the current state of art on SHA-1 collision attacks is entirely based on disturbance vectors for very compelling reasons, and published collision attacks only deviate from local collisions in the first 20 steps or the last 5 steps (75 up to 79), the current state of art solidly supports this conjecture with a safe large margin. Based on this conjecture, we show how we can efficiently determine such unavoidable conditions for the known cryptanalytic attack classes on SHA-1. Moreover, we show how we can exploit a significant overlap of unavoidable conditions between DVs that allows a more efficient checking of unavoidable bit conditions for many disturbance vectors simultaneously.

Collision detection uses *recompressions*, i.e., evaluations of the compression function starting from an intermediate state to uniquely determine the input and output chaining value for a given message block. Collision detection requires a recompression for each tested DV for each message block of a given message. Unavoidable bit conditions allow a significant improvement to collision detection by very quickly checking the unavoidable bit conditions per DV and only performing a recompression when all unavoidable bit conditions for that DV are satisfied.

We have implemented the improved SHA-1 collision detection using unavoidable conditions which checks 32 DVs (twice as many as previous work). The improved collision detection is 20 to 30 times faster than without our unavoidable condition improvements. We have measured that overall our improved SHA-1 collision detection is only a factor 1.60 slower on average than SHA-1. The correctness of our implementation follows from easily verified attack-class independent code, automatically generated tables for each attack class from a very short identification, and testing its correctness against the known SHA-1 collision.

After the demonstration of a SHA-1 collision, the open source implementation of our algorithms was included in Git. As part of incorporating our implementation, it was further optimized to meet Git's performance requirements. These performance improvements were small in comparison with the detection algorithm as it stood before our algorithmic optimizations, less than doubling the speed. However, these improvements made the difference between the algorithm being adopted in Git or not [32]. This shows that the more than 20 times improvement that unavoidable conditions introduce, in fact, make this algorithm usable in practice.

The remainder of our paper is organized as follows. In Sect. 3 we treat the formal concept of unavoidable conditions and their practical applications. How to determine them for known attack classes against SHA-1 and to maximize the overlap between the sets of unavoidable conditions between DVs is covered in Sect. 4. In Sect. 5 we disclose more specific details about our open-source implementation, in particular with regards how to efficiently check unavoidable bit conditions. We discuss performance aspects in Sect. 6.

## 3 Unavoidable conditions

### 3.1 Model

Necessary and/or sufficient bit conditions are a very useful tool for hash function cryptanalysis as laid out by Wang et al.[35]. In effect they reduce the problem of finding a message block pair that conforms to a differential path to the problem of finding a message block for which the bit conditions are satisfied. As well as reducing cost from computations over two compression function evaluations to only one compression function evaluation, such conditions allows more effective use of early stop techniques and advanced message modification techniques.

We define *unavoidable conditions* as conditions that are necessary for *all feasible attacks* in a certain attack class. While necessary and sufficient conditions for an attack can be easily and manually derived, determining unavoidable conditions is significantly harder as it requires the analysis of all feasible attacks in a certain attack class. We more formally define attack classes and such unavoidable conditions in a framework that we use to actually find unavoidable conditions for SHA-1 by showing these are necessary for all feasible attacks within an attack class.

Our attack class definition in Thm. 1 below is rather general but captures the functionality of many collision attacks variants (collision attack, pseudo-collision attack, near-collision attack) against compression functions: i.e., algorithms that output a pair of compression function inputs. Our general definition does not describe what the input or output differences should look like or, e.g., whether it requires specific values for $CV_1$ and $CV_2$. Instead such details are abstracted away as properties of specific attack classes.

**Definition 1** (Compression function attack class)**.** *For $N,M \in \mathbb{N}^+$, let $H : \{0,1\}^N \times \{0,1\}^M \to \{0,1\}^N$ be a compression function, then a class of attacks $\mathcal{C}$ against $H$ is a set of (randomized) algorithms $A$ that produce a* *tuple* $(CV_1, B_1, CV_2, B_2) \in \{0,1\}^N \times \{0,1\}^M \times \{0,1\}^N \times \{0,1\}^M$ *as output.*

We model an unavoidable condition for an attack class as a predicate over pairs $(CV, B)$ of a chaining value and message block. Such a predicate is called an unavoidable condition if and only if it holds for all possible $(CV_1, B_1)$ and $(CV_2, B_2)$ that may be output by any attack in the attack class.

**Definition 2** (Unavoidable condition)**.** *For $N, M \in \mathbb{N}^+$, let $H : \{0,1\}^N \times \{0,1\}^M \to \{0,1\}^N$ be a compression function and $\mathcal{C}$ be an attack class against $H$. Let $u : \{0,1\}^N \times \{0,1\}^M \to \{false, true\}$ be a non-trivial predicate over compression function inputs. Then $u$ is called an* unavoidable condition *for attack class $\mathcal{C}$ if and only if for all $A \in \mathcal{C}$ and for all possible outputs $(CV_1, B_1, CV_2, B_2) \leftarrow A$ it holds that $u(CV_1, B_1) = true$ and $u(CV_2, B_2) = true$.*

## 3.2 Speeding up collision detection

Let $\mathcal{S}$ be a set of attack classes. For each attack class $\mathcal{C} \in \mathcal{S}$ let $s_{\mathcal{C}} = (\delta B, i, \delta WS_i)$ be the associated message block difference, step $i$ and difference for the intermediate state after step $i$ as given in [23]. Also, let $\mathcal{U}_{\mathcal{C}}$ be a set of unavoidable conditions for each $\mathcal{C} \in \mathcal{S}$.

For each compression function evaluation during the hashing of a given message, collision detection will perform a recompression for every attack class $\mathcal{C} \in \mathcal{S}$. Such a recompression is rather costly as it results in that the overall cost of collision detection is a factor $|\mathcal{S}|$ more than only computing the hash.

If for compression function input $(CV, B)$ and for a given attack class $\mathcal{C}$ at least one unavoidable condition $u \in \mathcal{U}_{\mathcal{C}}$ is not satisfied then by definition $(CV, B)$ cannot be output by any attack $A \in \mathcal{C}$ (i.e., $(CV_1, B_1) = (CV, B)$ or $(CV_2, B_2) = (CV, B)$ as in Thm. 1). As an attack from $\mathcal{C}$ has been ruled out, a recompression for $\mathcal{C}$ is unnecessary and can be skipped. Alg. 1 is the improved collision detection that uses unavoidable conditions as preconditions before a performing a recompression. If the unavoidable conditions can be evaluated very quickly in comparison to the recompression, e.g., comparing whether two bits are equal/unequal in the internal state of the compression function, then a significant speed improvement can be achieved.

## 4 Application to SHA-1

### 4.1 Notation

SHA-1 is defined using 32-bit words $X = (x_i)_{i=0}^{31} \in \{0,1\}^{32}$ that are identified with elements $X = \sum_{i=0}^{31} x_i 2^i$ of $\mathbb{Z}/2^{32}\mathbb{Z}$ (for addition and subtraction). A *binary signed digit representation* (BSDR) for $X \in \mathbb{Z}/2^{32}\mathbb{Z}$ is a sequence $Z = (z_i)_{i=0}^{31} \in \{-1, 0, 1\}^{32}$ for which $X = \sum_{i=0}^{31} z_i 2^i$. We use the following notation: $Z[i] = z_i$, $RL(Z, n)$ and $RR(Z, n)$ (cyclic left and right rotation), $w(Z)$ (Hamming weight), $\sigma(Z) = X = \sum_{i=0}^{31} k_i 2^i \in \mathbb{Z}/2^{32}\mathbb{Z}$.

In collision attacks we consider two related messages $M$ and $M'$. For any variable $X$ related to the SHA-1 calculation of $M$, we use $X'$ to denote the corresponding variable for $M'$. Furthermore, for such a 'matched' variable $X \in \mathbb{Z}/2^{32}\mathbb{Z}$ we define $\delta X = X' - X$ and $\Delta X = (X'[i] - X[i])_{i=0}^{31}$.

### 4.2 SHA-1's compression function

The input for SHA-1's Compress consists of an intermediate hash value $CV_{in} = (a, b, c, d, e)$ of five 32-bit words and a 512-bit message block $B$. The 512-bit message block $B$ is partitioned into 16 consecutive 32-bit strings which are interpreted as 32-bit words $W_0, W_1, \ldots, W_{15}$ (using big-endian), and expanded to $W_0, \ldots, W_{79}$ as follows:

$$W_t = RL(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}, 1), \quad \text{for } 16 \le t < 80. \tag{1}$$

We describe SHA-1's compression function Compress in an 'unrolled' version. For each step $t = 0, \ldots, 79$ it uses a working state consisting of five 32-bit words $Q_t$, $Q_{t-1}$, $Q_{t-2}$, $Q_{t-3}$ and $Q_{t-4}$ and calculates a new state word $Q_{t+1}$. The working state is initialized before the first step as:

$$(Q_0, Q_{-1}, Q_{-2}, Q_{-3}, Q_{-4}) = (a, b, RR(c, 30), RR(d, 30), RR(e, 30)).$$

For $t = 0, 1, \ldots, 79$ in succession, $Q_{t+1}$ is calculated as follows:

$$\begin{aligned} F_t &= f_t(Q_{t-1}, RL(Q_{t-2}, 30), RL(Q_{t-3}, 30)), \\ Q_{t+1} &= F_t + AC_t + W_t + RL(Q_t, 5) + RL(Q_{t-4}, 30). \end{aligned} \tag{2}$$

These 80 steps are grouped in 4 rounds of 20 steps each. Here, $AC_t$ is the constant $\texttt{5a827999}_{16}$, $\texttt{6ed9eba1}_{16}$, $\texttt{8f1bbcdc}_{16}$ or $\texttt{ca62c1d6}_{16}$ for the 1st, 2nd, 3rd and 4th round, respectively. The non-linear function $f_t(X, Y, Z)$ is defined as $(X \wedge Y) \oplus (\overline{X} \wedge Z)$, $X \oplus Y \oplus Z$, $(X \wedge Y) \vee (Z \wedge (X \vee Y))$ or $X \oplus Y \oplus Z$ for the 1st, 2nd, 3rd and 4th round, respectively. Finally, the output intermediate hash value $CV_{out}$ is determined as:

$$CV_{out} = (a + Q_{80},\ b + Q_{79},\ c + RL(Q_{78}, 30),\ d + RL(Q_{77}, 30),\ e + RL(Q_{76}, 30)).$$

**Algorithm 1:** Improved collision detection

```
Let $H : \{0,1\}^N \times \{0,1\}^M \to \{0,1\}^N$, $IV \in \{0,1\}^N$ be an MD5-like compression function consisting of $I$
  reversible steps and a Davies-Meyer feed-forward.
Let $\mathcal{S}$ be a set of attack classes $s = (\delta B, i, \delta WS_i)$ and $\mathcal{U}_s$ a set of unavoidable conditions for each $s \in \mathcal{S}$.
The algorithm below returns True when a near-collision attack was detected and False otherwise.
Given padded message $P = P_1 || \dots || P_n$ consisting of $n$ blocks $P_j \in \{0,1\}^M$ do:
  1. Let $CV_0 = IV$ and do the following for $j = 1, \dots, n$:
      (a) Evaluate $CV_j = H(CV_{j-1}, P_j)$ and store intermediate working states $WS_i$ after each step $i = 0, \dots, I-1$ of $H$.
      (b) For each $s = (\delta B, i, \delta WS_i) \in \mathcal{S}$ do:
          i. If $u(CV_{j-1}, P_j) = false$ for some $u \in \mathcal{U}_s$ then skip steps ii.–vi.
          ii. Determine $P'_j = P_j + \delta B$, $WS'_i = WS_i + \delta WS_i$
          iii. Compute steps $i, i-1, \dots, 0$ of $H$ backwards to determine $CV'_{j-1}$
          iv. Compute steps $i+1, \dots, I-1$ forwards to determine $WS'_{I-1}$
          v. Determine $CV'_j$ from $CV'_{j-1}$ and $WS'_{I-1}$ (Davies-Meyer feed-forward)
          vi. If $CV'_j = CV_j$ return True
  2. Return False
```

## 4.3 Local collisions and the disturbance vector

In 1998, Chabaud and Joux [4] constructed a collision attack on SHA-0, SHA-1's withdrawn predecessor, based on local collisions. A local collision over 6 steps for SHA-0 and SHA-1 consists of a disturbance $\delta Q_{t+1} = 2^b$ created in some step $t$ by a message word bit difference $\delta W_t = 2^b$. This disturbance is corrected over the next five steps, so that after those five steps no differences occur in the five working state words. They were able to interleave many of these local collisions such that the message word differences $(\Delta W_t)_{t=0}^{79}$ conform to the message expansion (cf. Eq. 1). For more convenient analysis, they consider the *disturbance vector* which is a non-zero vector $(DV_t)_{t=0}^{79}$ conform to the message expansion where every '1'-bit $DV_t[b]$ marks the start of a local collision based on the disturbance $\delta W_t[b] = \pm 1$. We denote by $(DW_t)_{t=0}^{79}$ the message word bit differences without sign (i.e., $DW_t = W'_t \oplus W_t$) for a disturbance vector $(DV_t)_{t=0}^{79}$:

$$DW_t := \bigoplus_{(i,r)\in\mathcal{R}} RL(DV_{t-i}, r),$$

where

$$\mathcal{R} = \{(0,0),(1,5),(2,0),(3,30),(4,30),(5,30)\}.$$

Note that for each step one uses differences $\delta W_t$ instead of $DW_t$. We say that a message word difference $\delta W_t$ is *compatible* with $DW_t$ if there are coefficients $c_0, \dots, c_{31} \in \{-1,1\}$ such that $\delta W_t = \sum_{j=0}^{31} c_j \cdot DW_t[j]$. The set $\mathcal{W}_t$ of all compatible message word differences given $DW_t$ is defined as:

$$\mathcal{W}_t := \left\{ \sigma(X) \;\middle|\; \underset{X[i]\in\{-DW_t[i],+DW_t[i]\},\ i\in\{0,\dots,31\}}{\text{BSDR } X,} \right\} \qquad (3)$$

As for bit position 31 it holds that $-2^{31} \equiv 2^{31} \bmod 2^{32}$, only the signing of bits $0, \dots, 30$ affect the resulting $\delta W_t$. In fact for every $\delta W_t \in \mathcal{W}_t$ it holds that the coefficient $c_i \in \{-1,1\}$ for every bit position $i \in \{0, \dots, 30\}$ with $DW_t[i] = 1$ is uniquely determined.

## 4.4 Disturbance vector classes

Manuel [15] has classified previously found interesting disturbance vectors into two classes. A disturbance vector from the first class denoted by $\mathrm{I}(K,b)$ is defined by $DV_K = \dots = DV_{K+14} = 0$ and $DV_{K+15} = 2^b$. Similarly, a disturbance vector from the second class denoted by $\mathrm{II}(K,b)$ is defined by $DV_{K+1} = DV_{K+3} = RL(2^{31}, b)$ and $DV_{K+15} = 2^b$ and $DV_{K+i} = 0$ for $i \in \{0,2,4,5,\dots,14\}$. For both classes, the remaining $DV_0, \dots, DV_{K-1}$ and $DV_{K+16}, \dots, DV_{79}$ are determined through the (reverse) message expansion relation (Eq. 1).

## 4.5 Unavoidable conditions

The literature on collision attacks against SHA-1 (e.g., see [34, 21, 16, 11, 3, 17, 2, 37, 5, 36, 8, 15, 24]) consists entirely of attacks based on combinations of local collisions as prescribed by a disturbance vector. This is a common property and for a very compelling reason: it is the only known way to construct differential paths with message word differences compatible with the message expansion relation. Even then it seems that out of $2^{512}$ possible disturbance vectors there are only a few tens of disturbance vectors suitable for feasible cryptanalytic attacks.

In the first number of steps and the last few steps attacks can deviate from the DV-prescribed local collisions without a significant impact in the overall attack complexity. On the contrary, it is an important technique to use a specially crafted so-called 'non-linear' differential path for the first number of steps to allow arbitrary chaining value differences to be used in combination with the disturbance vector as introduced by Wang et al.[34]. Also, for the last few steps there may be higher probability differential steps as shown in [24]. However, deviating from DV-prescribed local collisions towards the middle becomes very costly very quickly as the resulting avalanche of perturbations directly leads to significant increases of the attack complexity. Hence, for the steps in the middle it remains unavoidable to use the DV-prescribed local collisions, which has led us to the following conjecture:

**Conjecture 3.** *Over steps* $[35,65)$ *it is unavoidable to use the DV-prescribed local collisions: deviating from the DV over these steps will result in an avalanche that will significantly increase the attack complexity.*

As published collision attacks only deviate from local collisions in the first 20 steps or the last 5 steps (75 up to 79) for reasons already mentioned, the current state of art solidly supports our conjecture with a safe margin. In fact we have considered taking a large range of steps in Thm. 3, however the increase in number of unavoidable conditions only results in a slight performance increase. In the end we opted for a larger safety margin instead of a slight performance increase.

Based on our Thm. 3, we propose to protect against attack classes based on disturbance vectors that use the prescribed local collisions over steps $[35,65)$. This restriction allows us to determine unavoidable conditions over all non-zero probability differential paths over steps 35 up to 65 that adhere to the disturbance vector. We propose to use unavoidable message bit relations that control the signs of bits in the $\Delta W_t$. These message bit relations are used in attacks to ensure that, e.g., adjacent active bits collapse to a single bit difference, or that two bits have opposing sign to cancel differences (the perturbation of each local collision). Looking at SHA-1 attacks, these message bit relations are all of the form $W_i[a] \oplus W_j[b] = c$ or $W_i[a] = c$, hence this specific form of unavoidable conditions can be checked very efficiently. But as noted before, one cannot simply use the necessary conditions of one attack, it is important to prove which of those message bit relations are necessary for *all* feasible attacks. We will refer to such unavoidable message bit relations as *unavoidable bit conditions* or *UBC*s. The method we can use to determine the UBCs for each disturbance vector is described below.

## 4.6 Using Joint-Local Collision Analysis

Choose any disturbance vector that may lead to a feasible collision attack. To determine the UBCs for this disturbance vector, we will need to work with the set of all possible DV-based differential paths over steps $[35,65)$. Any differential path uses fixed differences for each expanded message word, these directly imply values for some bits $W_t[i]$. The set of these bit positions $W_t[i]$ is independent of the differential path and is pre-determined by the DV. We map each differential path to a vector containing the values for these bit positions $W_t[i]$. Then we can look at the smallest affine vector space that encompasses all these vectors. This affine vector space can be represented by a system of linear equations over those message bits, which will directly give the desired unavoidable bit conditions. By construction it follows that any solution to any possible differential path based on this DV satisfies these unavoidable bit conditions. Therefore if an expanded message does not satisfy all UBCs then this message cannot be a solution for any possible differential path over steps $[35,65)$ based on this DV.

To efficiently compute UBCs we will use techniques introduced in [24] that allow efficient computations on large classes of differential paths that are otherwise not possible. We will present our method at a higher level using notation taken from [24]: Let $\mathcal{Q}_t$ be the set of all allowed state differences $\Delta Q_t$ given $(DV_i)_{i=0}^{79}$:

$$\mathcal{Q}_t := \left\{ \text{BSDR } Y \;\middle|\; {\scriptstyle \sigma(Y)=\sigma(Z), \atop Z[i]\in\{-DV_{t-1}[i],DV_{t-1}[i]\},\ i=0,\ldots,31} \right\}.$$

A differential path $\mathcal{P}$ over steps $t \in [35,65)$ is given as

$$\mathcal{P} = ((\Delta Q_t)_{t=35-4}^{64+1}, (\Delta F_t)_{t=35}^{64}, (\delta W_t)_{t=35}^{64}),$$

with correct differential steps for $t \in [35,65)$:

$$\begin{aligned} \sigma(\Delta Q_{t+1}) = \sigma(RL(\Delta Q_t,5)) + \sigma(RL(\Delta Q_{t-4},30)) \\ + \sigma(\Delta F_t) + \delta W_t. \end{aligned} \tag{4}$$

The success probability $\Pr[\mathcal{P}]$ of a differential path $\mathcal{P}$ is defined as the probability that the given path $\mathcal{P}$ holds exactly for uniformly-randomly chosen $\widehat{Q}_{35-4}, \ldots, \widehat{Q}_{35}$ and $\widehat{W}_{35}, \ldots, \widehat{W}_{64}$ and where the other variables are computed as defined in SHA-1's compression function. This can be efficiently computed (cf. [24]).

The set of *all* possible DV-based differential paths over steps $[35,65)$ that we will actually use to determine unavoidable bit conditions is defined as:

$$\mathcal{D}_{[35,65)} := \{\widehat{\mathcal{P}} \mid \Delta\widehat{Q}_i \in \mathcal{Q}_i,\ \delta\widehat{W_j} \in \mathcal{W}_j,\ \Pr[\widehat{\mathcal{P}}] > 0\}$$

Let $\mathcal{P} \in \mathcal{D}_{[35,65)}$ and let $\delta W_{35}, \ldots, \delta W_{64}$ be its message word differences. Let $t \in [35,65)$ and let $\mathcal{I}_t \subseteq \{0,\ldots,30\}$ be the set of bit positions $0 \leq i \leq 30$ such that $DW_t[i] = 1$. As $\delta W_t \in \mathcal{W}_t$, we have that $\delta W_t = \sum_{i=0}^{31} c_i \cdot DW_t[i]$ with $c_0, \ldots, c_{31} \in -1, 1$ (Eq. 3). We use the fact that the coefficients $c_i$ with $i \in \mathcal{I}_t$ are uniquely determined. This implies values for the bits $W_t[i]$ with $i \in \mathcal{I}_t$ as:

- if $c_i = 1$ then $\Delta W_t[i] = 1 \cdot DW_t = 1$ thus $W_t[i] = 0$ and $W_t'[i] = 1$;
- if $c_i = -1$ then $\Delta W_t[i] = -1 \cdot DW_t = -1$ thus $W_t[i] = 1$ and $W_t'[i] = 0$;

Hence, given $\mathcal{P} \in \mathcal{D}_{[35,65)}$ for $t \in [35,65)$ and $i \in \mathcal{I}_t$ the value of $W_t[i]$ is known. Let $X = ((t,i) \mid t \in [35,65) \wedge i \in \mathcal{I}_t)$ be a vector of all $(t,i)$ for which the value of $W_t[i]$ is known given $\mathcal{P} \in \mathcal{D}_{[35,65)}$ and let $R = |X|$ be the length of $X$. Then we can define a mapping that maps differential paths to a vector over $\mathbb{F}_2$ of the message bits $W_t[i]$ that are known:

$$\mu : \mathcal{D}_{[35,65)} \to \mathbb{F}_2^R : \quad \mathcal{P} \mapsto (W_t[i] | (t,i) = X[r])_{r=1}^R$$

And we can look at the smallest affine vector space $V$ that encapsulates the image $\mu(\mathcal{D}_{[35,65)})$ of $\mathcal{D}_{[35,65)}$. Although $V$ is uniquely determined, its representation $V = o + < v_1, \ldots, v_n >$ with an origin $o$ and generating vectors $v_1, \ldots, v_n$ is not unique. Let $\mathcal{P}_o \in \mathcal{D}_{[35,65)}$ be a fixed differential path, then we compute $V$ as:

$$o = \mu(\mathcal{P}_o), \quad \forall \mathcal{P} \in \mathcal{D}_{[35,65)} : v_{\mathcal{P}} = \mu(\mathcal{P}) - o.$$

Using linear algebra we can determine an equivalent description of $V$ as a system of equations over bits $W_t[i]$ with $(t,i) \in X$. This system of linear equations can be further manipulated using linear operations, and thus can be viewed as a linear space itself. So we use its 'row reduced form' which results entirely in equations over 2 message bits of the form $W_i[a] \oplus W_j[b] = c$.

For our improved SHA-1 collision detection implementation we have selected the 32 disturbance vectors with lowest estimated cost as in [24]. This is more than the 14 disturbance vectors intially suggested in [23], but using UBCs we could simply add protection against more DVs with very low extra cost. We ended up at 32 DVs as our UBC checking algorithm uses a 32-bit integer to hold a mask where each bit is associated with a DV and represents whether the UBCs of that DV are all fulfilled. The 32 disturbance vectors with number of UBCs in parentheses are given in Tbl. 1. The full listing of UBCs for these DVs is given in Appendix A.

## 4.7 Exploiting overlapping conditions between DVs

As disturbance vectors within each type I or II are all shifted and rotated versions of each other, disturbance vectors may have local collisions at the same positions and therefore may have some overlap in unavoidable bit conditions. In this section we try to maximize the number of UBCs shared between DVs by further manipulating the set of UBCs per DV. As each UBC is a linear equation, the set of UBCs per DV can be further manipulated for our purposes using simple linear operation.

In the previous section we analyzed 32 disturbance vectors and found 7 to 15 UBCs per DV with a total of 373 UBCs. The UBCs for each DV were generated in a 'row-reduced form' and this already leads to a significant overlap of UBCs: among the total of 373 UBCs there are only 263 distinct UBCs. E.g., UBC $W_{39}[4] \oplus W_{42}[29] = 0$ is shared among DVs I(45,0), I(49,0) and II(48,0). Using the procedure below we are able to reduce the number of distinct UBCs to 156. Note that for each DV the new set of UBCs remains equivalent to the original set of UBCs.

To minimize the overall amount of distinct UBCs we use a greedy selection algorithm to rebuild the set of UBCs per DV. Starting at an empty set of UBCs for each DV, our greedy algorithm in each step selects a new distinct UBC that is shared between as many DVs as possible and adds it to set of UBCs for the corresponding DVs. More specifically, for each DV it first generates a list of candidate UBCs by taking all linear combinations of the original set of UBCs and removes all candidates that are a linear combination of the current set of UBCs and thus that are already covered so far. Then it selects all UBCs that maximize the number of DVs it belongs to but is not covered so far. It rates each of those UBCs first based on weight (minimal weight prefered), second based on number of active bit positions (fewer bit positions prefered) and finally on the gap $j-i$ between the first $W_i$ and the last $W_j$ in the UBC. It selects the best rated UBC and adds that to UBC sets of the DVs it belongs to but is not covered so far. Finally, for each DV it will output a new set of UBCs that is equivalent to the original set of UBCs, but for which there are much fewer distinct UBCs over all DVs.

The output of improved sets of UBCs of our greedy selection algorithm for the 32 DVs and original 373 UBCs found in the previous section can be found in Appendix A. Using this approach we have further reduced the number of unique UBCs from 263 to 156, where each new UBC belongs up to 7 DVs.

In Sect. 5.1 we further comment on the implementation of this greedy algorithm that immediately outputs optimized C code for verifying UBCs for all 32 DVs simultaneously. This optimized C code is verified against a straightforward simple implementation using the original sets of 373 UBCs as described in Sect. 5.2.

Table 1: SHA-1 DV selection and number of UBCs

| | | | |
|---|---|---|---|
| I(43,0) (11 UBCs) | I(44,0) (12 UBCs) | I(45,0) (12 UBCs) | I(46,0) (11 UBCs) |
| I(46,2) (7 UBCs) | I(47,0) (12 UBCs) | I(47,2) (7 UBCs) | I(48,0) (14 UBCs) |
| I(48,2) (7 UBCs) | I(49,0) (13 UBCs) | I(49,2) (8 UBCs) | I(50,0) (14 UBCs) |
| I(50,2) (8 UBCs) | I(51,0) (15 UBCs) | I(51,2) (10 UBCs) | I(52,0) (14 UBCs) |
| II(45,0) (11 UBCs) | II(46,0) (11 UBCs) | II(46,2) (7 UBCs) | II(47,0) (14 UBCs) |
| II(48,0) (15 UBCs) | II(49,0) (14 UBCs) | II(49,2) (9 UBCs) | II(50,0) (14 UBCs) |
| II(50,2) (9 UBCs) | II(51,0) (14 UBCs) | II(51,2) (9 UBCs) | II(52,0) (15 UBCs) |
| II(53,0) (14 UBCs) | II(54,0) (14 UBCs) | II(55,0) (14 UBCs) | II(56,0) (14 UBCs) |

## 5 Implementation

This section describes the implementation of the UBC check in the SHA-1 Collision detection library. The source code for this library can be found at [28] This release contains the collision detection library that can be used in other software in the directory 'lib', the 'src' directory contains a modified sha1sum command line tool that uses the library. Both can be built by calling 'make' in the parent directory, additionally a special version 'sha1dcsum_partialcoll' is also included that specifically detects example collisions against reduced-round SHA-1 (as no full round SHA-1 collisions have been found yet.) Furthermore, in the directory 'tools' we provide the following:

- the original listing of UBCs per DV (directory 'data/3565');
- an example partial collision for SHA-1 (file 'test/sha1_reducedsha_coll.bin');
- the greedy selection algorithm from Sect. 4.7 that optimizes the UBC sets and outputs optimized code (directory 'parse_bitrel'), see Sect. 5.1;
- a program that verifies the optimized C code with optimized UBC sets against manually-verifiable C code (directory 'ubc_check_test'), see Sect. 5.2;

The collision detecting SHA-1 implementation, including the SHA-1 compression function as well as the collision detection logic and UBC checks, has been heavily optimized to be competitive with the performance of the prior implementation of SHA-1 in Git. This prior implementation of SHA-1 had been optimized in order to meet the performance requirements of the heavily utilized software. As such, we are assured that our implementation of the core SHA-1 functionality has been optimized to the point of being competitive with deployed and utilized implementations [32]. In Sect. 6 we discuss expected and measured performance of our improved SHA-1 collision detection.

### 5.1 Parse Bit Relations

This section describes the `parse_bitrel` program that implements the greedy selection algorithm described in Sect. 4.7 and generates source code for an optimized UBC check.

The greedy algorithm using the input UBC sets in directory 'data/3565' outputs improved UBC-sets for the DVs that have significant overlap. Another equivalent perspective is looking at the unique UBCs and the set of DVs each unique UBC belongs to, Appendix A lists the improved UBCs in this manner. The program - `parse_bitrel` uses this perspective to generate optimized source code for a function `ubc_check` which given an expanded message will return a mask of which DVs had all their UBCs satisfied.

As noted in Sect. 4.6 we have selected 32 disturbance vectors. Thus keeping track for which disturbance vectors a recompression is necessary conveniently fits in a 32 bit integer mask `C`. Each bit position in `C` will be associated with a particular DV $\mathrm{T}(k,b)$, where T represents the type I or II, and we have a named constant of the form `DV_T_K_B_bit` that will have only that bit set. Initially `C` will have all bits set and for each UBC that is not satisfied we will set bits to 0 at the bit positions of the DVs the UBC belongs to.

The UBCs for SHA-1 are of the form $W_i[a] \oplus W_j[b] = c$ as described in Sect. 4.6. The outcome of this condition is translated into a mask with all bits set or all bits cleared using the following C-code:

`M=0-(((W[i]>>a)^(W[j]>>b))&1)` if $c = 1$
`M=(((W[i]>>a)^(W[j]>>b))&1)-1` if $c = 0$

Note that in both of these cases, if UBC is satisfied then `M` results in a value with all bits set ($-1$ in 2's complement) and 0 otherwise.

Say the UBC belongs to multiple disturbance vectors `DV_T1_K1_B1_bit`, `DV_T2_K2_B2_bit`, ..., `DV_TN_KN_BN_bit`, then a mask is formed that has all other bits belonging to other DVs set to 0. This mask will be OR'ed into the mask `M` above to force bits to the value 1 for all bit positions associated with DVs not belonging to this unique UBC:

```
M | ~(DV_T1_K1_B1_bit | DV_T2_K2_B2_bit | ...
          | DV_TN_KN_BN_bit).
```

In effect, only the bit positions for DVs the unique UBC belongs to can be 0 which they will be if and only if the unique UBC is not satisfied. Hence, this last mask will be AND'ed into the variable C to conditionally clear the bits associated with these DVs if the UBC is not satisfied. For example, the following clause is one of the clauses generated by the parse_bitrel:

```
C &= ((((W[46]>>4)^(W[49]>>29))&1)-1) |
 ~( DV_I_46_0_bit | DV_I_48_0_bit | DV_I_50_0_bit |
    DV_I_52_0_bit | DV_II_50_0_bit | DV_II_55_0_bit );
```

The ubc_check function thus consists of initializing the variable C and statements for each unique UBC to update C as described above. The parse_bitrel program combines these clauses into a bit-wise AND of all the individual statements and generates the ubc_check function. The above example works for all cases. However, we can produce slightly better statements with fewer operations in certain cases which are omitted here, but can be found in the public source code.

## 5.2 UBCCheckTest: Correctness testing

This section describes the program ubc_check_test for correctness testing. The above program parse_bitrel will output optimized C-code for ubc_check that will verify all UBCs and output a mask whose bits mark whether a recompression for a particular DV is needed. For testing purposes one would like to have many test cases to run it on, however there are no SHA-1 example collisions at all. Hence, great care must be taken to ensure code correctness of the collision detection library. For this purpose we let parse_bitrel also output C-code for a function ubc_check_verify that will be equivalent to ubc_check but will be based on the original non-improved UBC-sets and use straightforward code that can be manually verified for correctness. After manual verification we know ubc_check_verify to be correct.

To ensure that ubc_check is correct we test its functional equivalence to the correct ubc_check_verify. As each individual UBC statement depends on only 2 expanded message bits $W_i[a]$ and $W_i[b]$, if an error exists it will trigger with probability at least 0.25 for random values. Unfortunately, such an error may be masked by other UBCs not being satisfied and forcing the bit positions in C with possible errors to 0 anyway. To ensure any error will reveal itself, we feed $2^{24}$ random inputs to both ubc_check and ubc_check_verify and verify whether their outputs are identical. As the highest number of UBCs of a DV is 15, if an error is located in the code of one of these UBCs we can still expect that out of the $2^{24}$ samples we will have approximately $2^{10}$ cases where all other UBCs for this DV are satisfied. In these cases the output bit for this DV of ubc_check and ubc_check_verify equals the output for the target UBC and the error will be exposed with probability at least 0.25 for each of these $2^{10}$ cases. The probability that an error with probability at least 0.25 will not occur in $2^{10}$ random inputs is at most $0.75^{1024} \approx 2^{-425}$. This, as well as a few other basic tests, ensures that our greedy selection algorithm for improved UBC-sets and the produced optimized C-code ubc_check contains no errors.

## 6 Performance

In this section we discuss the expected performance increase and we compare some measured speeds. We have compiled and tested the code on different compiler and processor technologies. The performance of the implementation was tested with several compilers, platforms and processors. For x86 performance the code was run on Linux, Windows and macOS. On Linux, the code was compiled for x86-64 with GCC 5.4.0 (gcc) and run on Ubuntu 16.04. The code was compiled for both x86-32 and x86-64 with the Microsoft Visual Studio 2015 C++ compiler (msvc) and run on Windows 10. In both of these cases, the code was run on an Intel Xeon L5520 running at 2.26GHz. For macOS, the code was compiled with Clang 4.2.1 (clang) targeting x86-64 macOS Sierra and run on an Intel Core i7 3615QM running at 2.30Ghz. To measure performance on ARM architecture, the code was compiled with GCC 4.9.2 (gcc arm) and run on a Raspberry Pi 3 running Raspbian Jessie with a quad-core Broadcom BCM2837, which is an ARM Cortex-A53, running at 1.2Ghz. Note that at the time of these experiments Raspbian Jessie runs in 32 bit mode only, even though this particular processor model can run in both 32 and 64 bit modes.

The performance numbers below vary a bit between different compiler and processor technologies due to different available processor instructions and different compiler optimizations. Such variances for a given platform could be eliminated using assembly code, however such code is very difficult to maintain and therefore not considered for our project, which intends to show the feasibility of these algorithms and techniques. Rather, assembler implementations of these algorithms can be considered by projects that require these collision detection and high performance implementations. Due to these variances the shown results should be taken as indicative speed improvements for other compilers and/or compiler optimization flags and/or processors.

Using UBCs, we will only do a recompress for a given DV if all its UBCs are satisfied. Let $\mathcal{S}$ be the set of DVs and $\mathcal{U}_{dv}$ be the set of UBCs for $dv \in \mathcal{S}$. Then the probability $p_{dv}$ that a random message block satisfies all UBCs associated with $dv \in \mathcal{S}$ is $p_{dv} = 2^{-|\mathcal{U}_{dv}|}$.

Table 2: Comparison of the performance of SHA-1's compression function and our `ubc_check` function. Units given in number of single message block operations per millisecond. `ubc_check` takes 46% to 76% of the time of `SHA1Compress`.

| | `SHA1Compress` | `ubc_check` |
|---|---|---|
| gcc x86-64 | 4236.38 | 8359.01(0.51×) |
| clang x86-64 | 5192.51 | 11363.47(0.46×) |
| msvc x86-64 | 2618.30 | 4445.76(0.59×) |
| msvc x86-32 | 2712.14 | 4663.56(0.58×) |
| gcc arm | 815.26 | 1074.47(0.76×) |

Hence, the expected cost of the recompressions for $dv \in \mathcal{S}$ is $p_{dv} \times n \times$ `SHA1Compress`, where $n$ is the number of message blocks for a given message, or equivalently $p_{dv} \times$ `SHA-1`.

The expected total cost of all recompressions for a given message of $n$ message blocks is therefore $(\sum_{dv \in \mathcal{S}} p_{dv}) \times$ `SHA-1`. For the 32 selected disturbance vectors given in Tbl. 1 together with their number of UBCs, we found that $\sum_{dv \in \mathcal{S}} p_{dv} \approx 0.0495$.

Therefore using UBCs we have reduced the cost of recompressions from $32 \times$ `SHA-1` to $\approx 0.0495 \times$ `SHA-1`, a speed improvement of a factor of about 646. Also, this implies that on average we can expect to do one recompression about every 20.2 message blocks. However, the total cost of collision detection includes the cost of SHA-1 as well as the cost of verifying the UBCs.

We have measured the cost of `ubc_check` in comparison to `SHA1Compress` in function calls per millisecond in Tbl. 2. These figures were determined by measuring the time of $2^{26}$ function calls on already prepared random inputs. The relative performance ratio `ubc_check`/ `SHA1Compress` is given in parentheses. We have measured that `ubc_check` takes about 46% to 76% of the time of `SHA1Compress` depending on the platform. Denote this ratio as $u$ then we can expect that the total cost of collision detection using UBCs is approximately $(1 + u + 0.0495) \times$ `SHA-1`. Hence, this leads to an estimated cost factor of about 1.51 to 1.81 of collision detection relative to the original SHA-1. Note that we expect the actual figures to be slightly lower as both the cost of the recompressions and the cost of `ubc_check` are expressed relatively to `SHA1Compress` and not to SHA-1 which actually includes some more overhead. This shows that the UBC check almost completely eliminates the amount of time doing full disturbance vector checks and the performance loss is purely spent by time in the `ubc_check` function itself. Thus using UBCs we expect collision detection to be possible in around three halves the time it takes to compute a single hash digest. Overall the relative timings of `ubc_check` shows that we can expect drastic speedups from using unavoidable conditions.

The analysis of the internal operations of the SHA-1 hash and collision detection ignores a great deal of overhead that the algorithm may incur. So it is necessary to do a more detailed performance analysis of the full collision detection algorithm. The scaling of this algorithm does not depend on the length of the input varying. So a reference timing for hashing random 2 kilobyte messages was used. This number was chosen because it is representative of the order of magnitude of bytes that must be hashed while verifying a single RSA certificate. Tbl. 3 shows the overall function calls per millisecond count for random 2KiB messages. We timed the original SHA-1 without collision detection, SHA-1 with collision detection with the UBC optimizations, and finally SHA-1 with collision detection but without using UBCs. The presented timings were determined by running the measured function on an already prepared random input in a loop with 512 iterations, and averaging these timings for 128 different random inputs. Note that these are preliminary performance numbers and have limited precision and more accurate numbers will be provided in later drafts of this paper.

As in the previous table the relative performance to SHA-1 is given in parentheses. For example, when compiled with gcc x86-64 the SHA-1 digest algorithm with hash collision detection but without the UBC check optimizations takes over 39 times the amount of time it takes to run the original digest algorithm. While adding the UBC check allows the collision detection code to run in well under double the time. This table shows that while adding the straight forward collision detection code increases the time of a SHA-1 computation by around 30 to 40 times, using the UBC check optimizations allows a SHA-1 computation with collision detection to be run in about 1.6× the time.

## 7 Future directions

From our results it is clear unavoidable conditions can be used for a significant speed up for collision detection resulting in only a small performance loss compared to the performance of the original cryptographic primitive SHA-1.

We intend to supply additional reference code to ease use of our SHA-1 collision detection library in all application that use OpenSSL [30] in future work. This should make collision detection significantly easier to apply in applications even for developers with limited experience with OpenSSL and cryptographic libraries.

Another future research direction is how to determine unavoidable conditions for MD5. MD5 is significantly weaker than SHA-1 and there are 223 known at-

Table 3: Performance numbers for message block computations of the SHA-1 Message Digest algorithm, units given in number of 2KiB messages hashed per millisecond. Collision detection using UBCs is 1.43 to 1.66 times slower than SHA-1, however without using UBCs collision detection is 30 to 43 times slower than SHA-1.

| | SHA1 | SHA1DC no UBC Check | SHA1DC UBC Check |
|---|---|---|---|
| gcc x86-64 | 148.14 | 3.75(39.50×) | 92.82(1.60×) |
| clang x86-64 | 226.60 | 7.58(29.88×) | 136.33(1.66×) |
| msvc x86-64 | 115.80 | 2.69(42.98×) | 72.23(1.60×) |
| msvc x86-32 | 83.42 | 2.06(40.58×) | 58.14(1.43×) |
| gcc arm | 26.11 | 0.81(32.04×) | 16.30(1.60×) |

tack classes that are based on a number of different approaches to construct a high probability differential path over the most important steps that contribute to the complexity. It is thus significantly more challenging to find UBCs for these classes and will require a more close study of the different main approaches. Nevertheless, as MD5 collision detection is 224 times slower than MD5, there is ample room and demand for speed improvements.

## 8 Conclusion

In this paper we have presented a significant performance improvement for collision detection, which is very timely due to the recently announced first collision for SHA-1. We have formally treated a new concept of unavoidable conditions that the output of any feasible attack in an attack class must satisfy. Furthermore, based on a conjecture solidly supported by the current state of the art, we have shown how we can determine unavoidable bit conditions (UBC) for SHA-1 and how to maximize the overlap between the UBC sets of different DVs. We have implemented the improved SHA-1 collision detection using such unavoidable conditions and which is about 20 to 30 times faster than without our unavoidable condition improvements. We have measured that overall our implementation of SHA-1 with collision detection is only a factor 1.60 slower on average than the original SHA-1.

That collisions attacks are a realistic and significant threat is very clear given the rogue Certification Authority publication [29] and the exposed Windows Update signature forgery in the supermalware Flame [23]. This shows that nation states have the resources to carry out such attacks and exploit them in the real world. Furthermore SHA-1-based signatures are still used at large and are also supported for verification almost ubiquitously. More protection against signature forgeries is greatly warranted and our improved SHA-1 collision detection enables protection against digital signature forgeries at a very low cost.

As SHA-1 is practically broken, yet SHA-1-based signatures are still used at large and are also widely supported (at least for verification), our improved SHA-1 collision detection enables protection against digital signature forgeries at a very low cost. Our improved implementation was deemed effective enough for Git, GitHub, Google Drive, Gmail and others to already deploy it in practice.

## Acknowledgments

The authors would like to acknowledge the code review feedback given by the developers in the Git community, which has greatly improved the quality of our implementation. The optimization suggestions given by Linus Torvalds and Jeff King (peff) especially significantly improved performance. The authors would also like to thank the anonymous reviewers who took time to give detailed feedback and suggestions for improving this paper.

## References

[1] Gilles Brassard (ed.), *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, Lecture Notes in Computer Science, vol. 435, Springer, 1990.

[2] Christophe De Cannière, Florian Mendel, and Christian Rechberger, *Collisions for 70-Step SHA-1: On the Full Cost of Collision Search*, Selected Areas in Cryptography (Carlisle M. Adams, Ali Miri, and Michael J. Wiener, eds.), Lecture Notes in Computer Science, vol. 4876, Springer, 2007, pp. 56–73.

[3] Christophe De Cannière and Christian Rechberger, *Finding SHA-1 Characteristics: General Results and Applications*, ASIACRYPT (Xuejia Lai and Kefei Chen, eds.), Lecture Notes in Computer Science, vol. 4284, Springer, 2006, pp. 1–20.

[4] Florent Chabaud and Antoine Joux, *Differential Collisions in SHA-0*, CRYPTO (Hugo Krawczyk, ed.), Lecture Notes in Computer Science, vol. 1462, Springer, 1998, pp. 56–71.

[5] Martin Cochran, *Notes on the Wang et al.* $2^{63}$ *SHA-1 Differential Path*, Cryptology ePrint Archive, Report 2007/474, 2007.

[6] Ivan Damgård, *A Design Principle for Hash Functions*, in Brassard [1], pp. 416–427.

[7] Max Fillinger and Marc Stevens, *Reverse-Engineering of the Cryptanalytic Attack Used in the Flame Super-Malware*, ASIACRYPT (Tetsu Iwata and Jung Hee Cheon, eds.), Lecture Notes in Computer Science, vol. 9453, Springer, 2015, pp. 586–611.

[8] E.A. Grechnikov, *Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics*, Cryptology ePrint Archive, Report 2010/413, 2010.

[9] E.A. Grechnikov and A.V. Adinetz, *Collision for 75-step SHA-1: Intensive Parallelization with GPU*, Cryptology ePrint Archive, Report 2011/641, 2011, `http://eprint.iacr.org/2011/641`.

[10] InfoWorld, *Oracle to Java devs: Stop signing JAR files with MD5*, January 2017.

[11] Charanjit S. Jutla and Anindya C. Patthak, *A Matching Lower Bound on the Minimum Weight of SHA-1 Expansion Code*, Cryptology ePrint Archive, Report 2005/266, 2005.

[12] Pierre Karpman, Thomas Peyrin, and Marc Stevens, *Practical Free-Start Collision Attacks on 76-step SHA-1*, CRYPTO (Rosario Gennaro and Matthew Robshaw, eds.), Lecture Notes in Computer Science, vol. 9215, Springer, 2015, pp. 623–642.

[13] CrySyS Lab, *sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks*, Laboratory of Cryptography and System Security, Budapest University of Technology and Economics, May 31, 2012.

[14] Kaspersky Lab, *The Flame: Questions and Answers*, Securelist blog, May 28, 2012.

[15] Stéphane Manuel, *Classification and generation of disturbance vectors for collision attacks against SHA-1*, Des. Codes Cryptography **59** (2011), no. 1-3, 247–263.

[16] Krystian Matusiewicz and Josef Pieprzyk, *Finding Good Differential Patterns for Attacks on SHA-1*, WCC (Øyvind Ytrehus, ed.), Lecture Notes in Computer Science, vol. 3969, Springer, 2005, pp. 164–177.

[17] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen, *The Impact of Carries on the Complexity of Collision Attacks on SHA-1*, FSE (Matthew J. B. Robshaw, ed.), Lecture Notes in Computer Science, vol. 4047, Springer, 2006, pp. 278–292.

[18] Ralph C. Merkle, *One Way Hash Functions and DES*, in Brassard [1], pp. 428–446.

[19] National Institute of Standards and Technology NIST, *FIPS PUB 180-1: Secure Hash Standard*, 1995.

[20] The Washington Post, *U.S., Israel developed Flame computer virus to slow Iranian nuclear efforts, officials say*, Ellen Nakashima, Greg Miller and Julie Tate, June 2012.

[21] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen, *Exploiting Coding Theory for Collision Attacks on SHA-1*, IMA Int. Conf. (Nigel P. Smart, ed.), Lecture Notes in Computer Science, vol. 3796, Springer, 2005, pp. 78–95.

[22] Amazon Web Services, *Amazon EC2 – Virtual Server Hosting*, aws.amazon.com, Retrieved Jan. 2016.

[23] Marc Stevens, *Counter-Cryptanalysis*, CRYPTO (Ran Canetti and Juan A. Garay, eds.), Lecture Notes in Computer Science, vol. 8042-I, Springer, 2013, pp. 129–146.

[24] Marc Stevens, *New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis*, EUROCRYPT (Thomas Johansson and Phong Q. Nguyen, eds.), Lecture Notes in Computer Science, vol. 7881, Springer, 2013, pp. 245–261.

[25] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov, *SHAttered*, February 2017, `http://shattered.io`.

[26] Marc Stevens, Pierre Karpman, and Thomas Peyrin, *Freestart Collision for Full SHA-1*, EUROCRYPT (Marc Fischlin and Jean-Sébastien Coron, eds.), Lecture Notes in Computer Science, vol. 9665, Springer, 2016, pp. 459–483.

[27] Marc Stevens, Arjen K. Lenstra, and Benne de Weger, *Chosen-Prefix Collisions for MD5 and*

*Colliding X.509 Certificates for Different Identities*, EUROCRYPT (Moni Naor, ed.), Lecture Notes in Computer Science, vol. 4515, Springer, 2007, pp. 1–22.

[28] Marc Stevens and Dan Shumow, *sha1collisiondetection*, GitHub, 2017.

[29] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger, *Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate*, CRYPTO (Shai Halevi, ed.), Lecture Notes in Computer Science, vol. 5677, Springer, 2009, pp. 55–69.

[30] The OpenSSL Project, *OpenSSL: The Open Source toolkit for SSL/TLS*, 1998, `www.openssl.org`.

[31] ThreadPost, *SHA-1 end times have arrived*, January 2017.

[32] Linus Torvalds, *Put sha1dc on a diet*, Mar 2017.

[33] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, Report 2004/199, 2004.

[34] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, *Finding Collisions in the Full SHA-1*, CRYPTO (Victor Shoup, ed.), Lecture Notes in Computer Science, vol. 3621, Springer, 2005, pp. 17–36.

[35] Xiaoyun Wang and Hongbo Yu, *How to Break MD5 and Other Hash Functions*, EUROCRYPT (Ronald Cramer, ed.), Lecture Notes in Computer Science, vol. 3494, Springer, 2005, pp. 19–35.

[36] Jun Yajima, Terutoshi Iwasaki, Yusuke Naito, Yu Sasaki, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta, *A strict evaluation method on the number of conditions for the SHA-1 collision search*, ASIACCS (Masayuki Abe and Virgil D. Gligor, eds.), ACM, 2008, pp. 10–20.

[37] Jun Yajima, Yu Sasaki, Yusuke Naito, Terutoshi Iwasaki, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta, *A New Strategy for Finding a Differential Path of SHA-1*, ACISP (Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, eds.), Lecture Notes in Computer Science, vol. 4586, Springer, 2007, pp. 45–58.

## A Unavoidable bit conditions

The tables below list the UBCs we have found in Sect. 4.6 and after processing to exploit their overlap as in Sect. 4.7. Instead of listing DVs with their UBCs, we list the UBCs together with the list of DVs they belong to.

Table 4: Overlapping unavoidable bit conditions

| UBC | List of DVs the UBC belongs to |
|---|---|
| $W_{44}[29] \oplus W_{45}[29] = 0$ | I(48,0) I(51,0) I(52,0) II(45,0) II(46,0) II(50,0) II(51,0) |
| $W_{49}[29] \oplus W_{50}[29] = 0$ | I(46,0) II(45,0) II(50,0) II(51,0) II(55,0) II(56,0) |
| $W_{48}[29] \oplus W_{49}[29] = 0$ | I(45,0) I(52,0) II(49,0) II(50,0) II(54,0) II(55,0) |
| $W_{47}[29] \oplus W_{48}[29] = 0$ | I(44,0) I(51,0) II(48,0) II(49,0) II(53,0) II(54,0) |
| $W_{46}[29] \oplus W_{47}[29] = 0$ | I(43,0) I(50,0) II(47,0) II(48,0) II(52,0) II(53,0) |
| $W_{45}[29] \oplus W_{46}[29] = 0$ | I(49,0) I(52,0) II(46,0) II(47,0) II(51,0) II(52,0) |
| $W_{43}[29] \oplus W_{44}[29] = 0$ | I(47,0) I(50,0) I(51,0) II(45,0) II(49,0) II(50,0) |
| $W_{40}[29] \oplus W_{41}[29] = 0$ | I(44,0) I(47,0) I(48,0) II(46,0) II(47,0) II(56,0) |
| $W_{47}[4] \oplus W_{50}[29] = 0$ | I(47,0) I(49,0) I(51,0) II(45,0) II(51,0) II(56,0) |
| $W_{46}[4] \oplus W_{49}[29] = 0$ | I(46,0) I(48,0) I(50,0) I(52,0) II(50,0) II(55,0) |
| $W_{45}[4] \oplus W_{48}[29] = 0$ | I(45,0) I(47,0) I(49,0) I(51,0) II(49,0) II(54,0) |
| $W_{44}[4] \oplus W_{47}[29] = 0$ | I(44,0) I(46,0) I(48,0) I(50,0) II(48,0) II(53,0) |
| $W_{43}[4] \oplus W_{46}[29] = 0$ | I(43,0) I(45,0) I(47,0) I(49,0) II(47,0) II(52,0) |
| $W_{42}[4] \oplus W_{45}[29] = 0$ | I(44,0) I(46,0) I(48,0) I(52,0) II(46,0) II(51,0) |
| $W_{41}[4] \oplus W_{44}[29] = 0$ | I(43,0) I(45,0) I(47,0) I(51,0) II(45,0) II(50,0) |
| $W_{54}[29] \oplus W_{55}[29] = 0$ | I(51,0) II(47,0) II(50,0) II(55,0) II(56,0) |
| $W_{53}[29] \oplus W_{54}[29] = 0$ | I(50,0) II(46,0) II(49,0) II(54,0) II(55,0) |
| $W_{52}[29] \oplus W_{53}[29] = 0$ | I(49,0) II(45,0) II(48,0) II(53,0) II(54,0) |
| $W_{50}[29] \oplus W_{51}[29] = 0$ | I(47,0) II(46,0) II(51,0) II(52,0) II(56,0) |
| $W_{42}[29] \oplus W_{43}[29] = 0$ | I(46,0) I(49,0) I(50,0) II(48,0) II(49,0) |
| $W_{41}[29] \oplus W_{42}[29] = 0$ | I(45,0) I(48,0) I(49,0) II(47,0) II(48,0) |
| $W_{50}[4] \oplus W_{53}[29] = 0$ | I(50,0) I(52,0) II(46,0) II(48,0) II(54,0) |
| $W_{49}[4] \oplus W_{52}[29] = 0$ | I(49,0) I(51,0) II(45,0) II(47,0) II(53,0) |
| $W_{48}[4] \oplus W_{51}[29] = 0$ | I(48,0) I(50,0) I(52,0) II(46,0) II(52,0) |
| $W_{40}[4] \oplus W_{43}[29] = 0$ | I(44,0) I(46,0) I(50,0) II(49,0) II(56,0) |
| $W_{39}[4] \oplus W_{42}[29] = 0$ | I(43,0) I(45,0) I(49,0) II(48,0) II(55,0) |
| $W_{38}[4] \oplus W_{41}[29] = 0$ | I(44,0) I(48,0) II(47,0) II(54,0) II(56,0) |
| $W_{37}[4] \oplus W_{40}[29] = 0$ | I(43,0) I(47,0) II(46,0) II(53,0) II(55,0) |
| $W_{55}[29] \oplus W_{56}[29] = 0$ | I(52,0) II(48,0) II(51,0) II(56,0) |
| $W_{51}[29] \oplus W_{52}[29] = 0$ | I(48,0) II(47,0) II(52,0) II(53,0) |
| $W_{52}[4] \oplus W_{55}[29] = 0$ | I(52,0) II(48,0) II(50,0) II(56,0) |
| $W_{51}[4] \oplus W_{54}[29] = 0$ | I(51,0) II(47,0) II(49,0) II(55,0) |
| $W_{36}[4] \oplus W_{40}[29] = 0$ | I(46,0) I(49,0) II(45,0) II(48,0) |
| $W_{45}[6] \oplus W_{47}[6] = 0$ | I(47,2) I(49,2) I(51,2) |
| $W_{44}[6] \oplus W_{46}[6] = 0$ | I(46,2) I(48,2) I(50,2) |
| $W_{35}[4] \oplus W_{39}[29] = 0$ | I(45,0) I(48,0) II(47,0) |
| $W_{53}[29] \oplus W_{56}[29] = 1$ | I(52,0) II(48,0) II(49,0) |
| $W_{51}[29] \oplus W_{54}[29] = 1$ | I(50,0) II(46,0) II(47,0) |
| $W_{50}[29] \oplus W_{52}[29] = 1$ | I(49,0) I(51,0) II(45,0) |
| $W_{49}[29] \oplus W_{51}[29] = 1$ | I(48,0) I(50,0) I(52,0) |
| $W_{48}[29] \oplus W_{50}[29] = 1$ | I(47,0) I(49,0) I(51,0) |
| $W_{47}[29] \oplus W_{49}[29] = 1$ | I(46,0) I(48,0) I(50,0) |
| $W_{46}[29] \oplus W_{48}[29] = 1$ | I(45,0) I(47,0) I(49,0) |
| $W_{45}[29] \oplus W_{47}[29] = 1$ | I(44,0) I(46,0) I(48,0) |
| $W_{44}[29] \oplus W_{46}[29] = 1$ | I(43,0) I(45,0) I(47,0) |
| $W_{40}[4] \oplus W_{42}[4] = 1$ | I(44,0) I(46,0) II(56,0) |
| $W_{39}[4] \oplus W_{41}[4] = 1$ | I(43,0) I(45,0) II(55,0) |

| | |
|---|---|
| $W_{38}[4] \oplus W_{40}[4] = 1$ | I(44,0) II(54,0) II(56,0) |
| $W_{37}[4] \oplus W_{39}[4] = 1$ | I(43,0) II(53,0) II(55,0) |
| $W_{41}[1] \oplus W_{42}[6] = 1$ | I(48,2) II(46,2) II(51,2) |
| $W_{40}[1] \oplus W_{41}[6] = 1$ | I(47,2) I(51,2) II(50,2) |
| $W_{39}[1] \oplus W_{40}[6] = 1$ | I(46,2) I(50,2) II(49,2) |
| $W_{36}[1] \oplus W_{37}[6] = 1$ | I(47,2) I(50,2) II(46,2) |
| $W_{58}[29] \oplus W_{59}[29] = 0$ | II(51,0) II(54,0) |
| $W_{57}[29] \oplus W_{58}[29] = 0$ | II(50,0) II(53,0) |
| $W_{56}[29] \oplus W_{57}[29] = 0$ | II(49,0) II(52,0) |
| $W_{48}[6] \oplus W_{50}[6] = 0$ | I(50,2) II(46,2) |
| $W_{47}[6] \oplus W_{49}[6] = 0$ | I(49,2) I(51,2) |
| $W_{46}[6] \oplus W_{48}[6] = 0$ | I(48,2) I(50,2) |
| $W_{43}[6] \oplus W_{45}[6] = 0$ | I(47,2) I(49,2) |
| $W_{42}[6] \oplus W_{44}[6] = 0$ | I(46,2) I(48,2) |
| $W_{50}[6] \oplus W_{51}[1] = 0$ | I(50,2) II(46,2) |
| $W_{47}[6] \oplus W_{48}[1] = 0$ | I(47,2) II(51,2) |
| $W_{46}[6] \oplus W_{47}[1] = 0$ | I(46,2) II(50,2) |
| $W_{42}[6] \oplus W_{43}[1] = 0$ | II(46,2) II(51,2) |
| $W_{41}[6] \oplus W_{42}[1] = 0$ | I(51,2) II(50,2) |
| $W_{40}[6] \oplus W_{41}[1] = 0$ | I(50,2) II(49,2) |
| $W_{56}[4] \oplus W_{59}[29] = 0$ | II(52,0) II(54,0) |
| $W_{55}[4] \oplus W_{58}[29] = 0$ | II(51,0) II(53,0) |
| $W_{54}[4] \oplus W_{57}[29] = 0$ | II(50,0) II(52,0) |
| $W_{53}[4] \oplus W_{56}[29] = 0$ | II(49,0) II(51,0) |
| $W_{39}[4] \oplus W_{43}[29] = 0$ | I(52,0) II(51,0) |
| $W_{38}[4] \oplus W_{42}[29] = 0$ | I(51,0) II(50,0) |
| $W_{37}[4] \oplus W_{41}[29] = 0$ | I(50,0) II(49,0) |
| $W_{35}[3] \oplus W_{39}[28] = 0$ | I(51,0) II(47,0) |
| $W_{63}[0] \oplus W_{64}[5] = 1$ | I(48,0) II(48,0) |
| $W_{62}[0] \oplus W_{63}[5] = 1$ | I(47,0) II(47,0) |
| $W_{61}[0] \oplus W_{62}[5] = 1$ | I(46,0) II(46,0) |
| $W_{60}[0] \oplus W_{61}[5] = 1$ | I(45,0) II(45,0) |
| $W_{56}[29] \oplus W_{59}[29] = 1$ | II(51,0) II(52,0) |
| $W_{48}[29] \oplus W_{55}[29] = 1$ | I(51,0) I(52,0) |
| $W_{36}[4] \oplus W_{38}[4] = 1$ | II(52,0) II(54,0) |
| $W_{63}[1] \oplus W_{64}[6] = 1$ | I(45,0) II(45,0) |
| $W_{61}[2] \oplus W_{62}[7] = 1$ | I(46,2) II(46,2) |
| $W_{44}[1] \oplus W_{45}[6] = 1$ | I(51,2) II(49,2) |
| $W_{37}[1] \oplus W_{38}[6] = 1$ | I(48,2) I(51,2) |
| $W_{35}[1] \oplus W_{36}[6] = 1$ | I(46,2) I(49,2) |

Table 5: Remaining unavoidable bit conditions

| UBC | DV of UBC | UBC | DV of UBC |
|---|---|---|---|
| $W_{59}[29] \oplus W_{60}[29] = 0$ | II(52,0) | $W_{53}[6] \oplus W_{55}[6] = 0$ | II(51,2) |
| $W_{52}[6] \oplus W_{54}[6] = 0$ | II(50,2) | $W_{51}[6] \oplus W_{53}[6] = 0$ | II(49,2) |
| $W_{49}[6] \oplus W_{51}[6] = 0$ | I(51,2) | $W_{41}[6] \oplus W_{43}[6] = 0$ | I(47,2) |
| $W_{40}[6] \oplus W_{42}[6] = 0$ | I(46,2) | $W_{37}[1] \oplus W_{37}[6] = 0$ | I(51,2) |
| $W_{55}[6] \oplus W_{56}[1] = 0$ | II(51,2) | $W_{54}[6] \oplus W_{55}[1] = 0$ | II(50,2) |
| $W_{53}[6] \oplus W_{54}[1] = 0$ | II(49,2) | $W_{51}[6] \oplus W_{52}[1] = 0$ | I(51,2) |
| $W_{49}[6] \oplus W_{50}[1] = 0$ | I(49,2) | $W_{48}[6] \oplus W_{49}[1] = 0$ | I(48,2) |
| $W_{45}[6] \oplus W_{46}[1] = 0$ | II(49,2) | $W_{39}[6] \oplus W_{40}[1] = 0$ | I(49,2) |
| $W_{57}[4] \oplus W_{59}[29] = 0$ | II(55,0) | $W_{60}[4] \oplus W_{64}[29] = 0$ | II(56,0) |

| | | | |
|---|---|---|---|
| $W_{60}[5] \oplus W_{64}[30] = 0$ | I(44,0) | $W_{59}[4] \oplus W_{63}[29] = 0$ | II(55,0) |
| $W_{59}[5] \oplus W_{63}[30] = 0$ | I(43,0) | $W_{58}[4] \oplus W_{62}[29] = 0$ | II(54,0) |
| $W_{57}[4] \oplus W_{61}[29] = 0$ | II(53,0) | $W_{44}[3] \oplus W_{48}[28] = 0$ | II(56,0) |
| $W_{44}[4] \oplus W_{48}[29] = 0$ | II(56,0) | $W_{43}[3] \oplus W_{47}[28] = 0$ | II(55,0) |
| $W_{43}[4] \oplus W_{47}[29] = 0$ | II(55,0) | $W_{42}[3] \oplus W_{46}[28] = 0$ | II(54,0) |
| $W_{42}[4] \oplus W_{46}[29] = 0$ | II(54,0) | $W_{41}[3] \oplus W_{45}[28] = 0$ | II(53,0) |
| $W_{41}[4] \oplus W_{45}[29] = 0$ | II(53,0) | $W_{40}[3] \oplus W_{44}[28] = 0$ | II(52,0) |
| $W_{40}[4] \oplus W_{44}[29] = 0$ | II(52,0) | $W_{39}[3] \oplus W_{43}[28] = 0$ | II(51,0) |
| $W_{39}[5] \oplus W_{43}[30] = 0$ | II(51,2) | $W_{38}[3] \oplus W_{42}[28] = 0$ | II(50,0) |
| $W_{38}[5] \oplus W_{42}[30] = 0$ | II(50,2) | $W_{37}[3] \oplus W_{41}[28] = 0$ | II(49,0) |
| $W_{37}[5] \oplus W_{41}[30] = 0$ | II(49,2) | $W_{36}[3] \oplus W_{40}[28] = 0$ | II(48,0) |
| $W_{35}[5] \oplus W_{39}[30] = 0$ | I(51,2) | $W_{59}[0] \oplus W_{64}[30] = 1$ | I(44,0) |
| $W_{58}[0] \oplus W_{63}[30] = 1$ | I(43,0) | $W_{58}[29] \oplus W_{61}[29] = 1$ | II(53,0) |
| $W_{55}[29] \oplus W_{58}[29] = 1$ | II(50,0) | $W_{52}[1] \oplus W_{56}[1] = 1$ | II(51,2) |
| $W_{51}[1] \oplus W_{55}[1] = 1$ | II(50,2) | $W_{50}[1] \oplus W_{54}[1] = 1$ | II(49,2) |
| $W_{47}[1] \oplus W_{51}[1] = 1$ | II(46,2) | $W_{46}[1] \oplus W_{48}[1] = 1$ | II(51,2) |
| $W_{45}[1] \oplus W_{47}[1] = 1$ | II(50,2) | $W_{43}[1] \oplus W_{51}[1] = 1$ | I(50,2) |
| $W_{42}[1] \oplus W_{50}[1] = 1$ | I(49,2) | $W_{38}[0] \oplus W_{43}[30] = 1$ | II(51,2) |
| $W_{38}[1] \oplus W_{40}[1] = 1$ | I(49,2) | $W_{38}[4] \oplus W_{39}[4] = 1$ | I(52,0) |
| $W_{37}[0] \oplus W_{42}[30] = 1$ | II(50,2) | $W_{37}[4] \oplus W_{38}[4] = 1$ | I(51,0) |
| $W_{36}[0] \oplus W_{41}[30] = 1$ | II(49,2) | $W_{36}[4] \oplus W_{37}[4] = 1$ | I(50,0) |
| $W_{63}[2] \oplus W_{64}[7] = 1$ | I(48,2) | $W_{62}[1] \oplus W_{63}[6] = 1$ | I(44,0) |
| $W_{62}[2] \oplus W_{63}[7] = 1$ | I(47,2) | $W_{61}[1] \oplus W_{62}[6] = 1$ | I(43,0) |
| $W_{39}[30] \oplus W_{44}[28] = 1$ | II(52,0) | $W_{38}[30] \oplus W_{43}[28] = 1$ | II(51,0) |
| $W_{37}[30] \oplus W_{42}[28] = 1$ | II(50,0) | $W_{36}[30] \oplus W_{41}[28] = 1$ | II(49,0) |
| $W_{35}[30] \oplus W_{40}[28] = 1$ | II(48,0) | | |