SHAttered: SHA-1 Collision for the (GPU-packing) Masses

Ben Prather
Algorithms Interest Group, April 4 2017

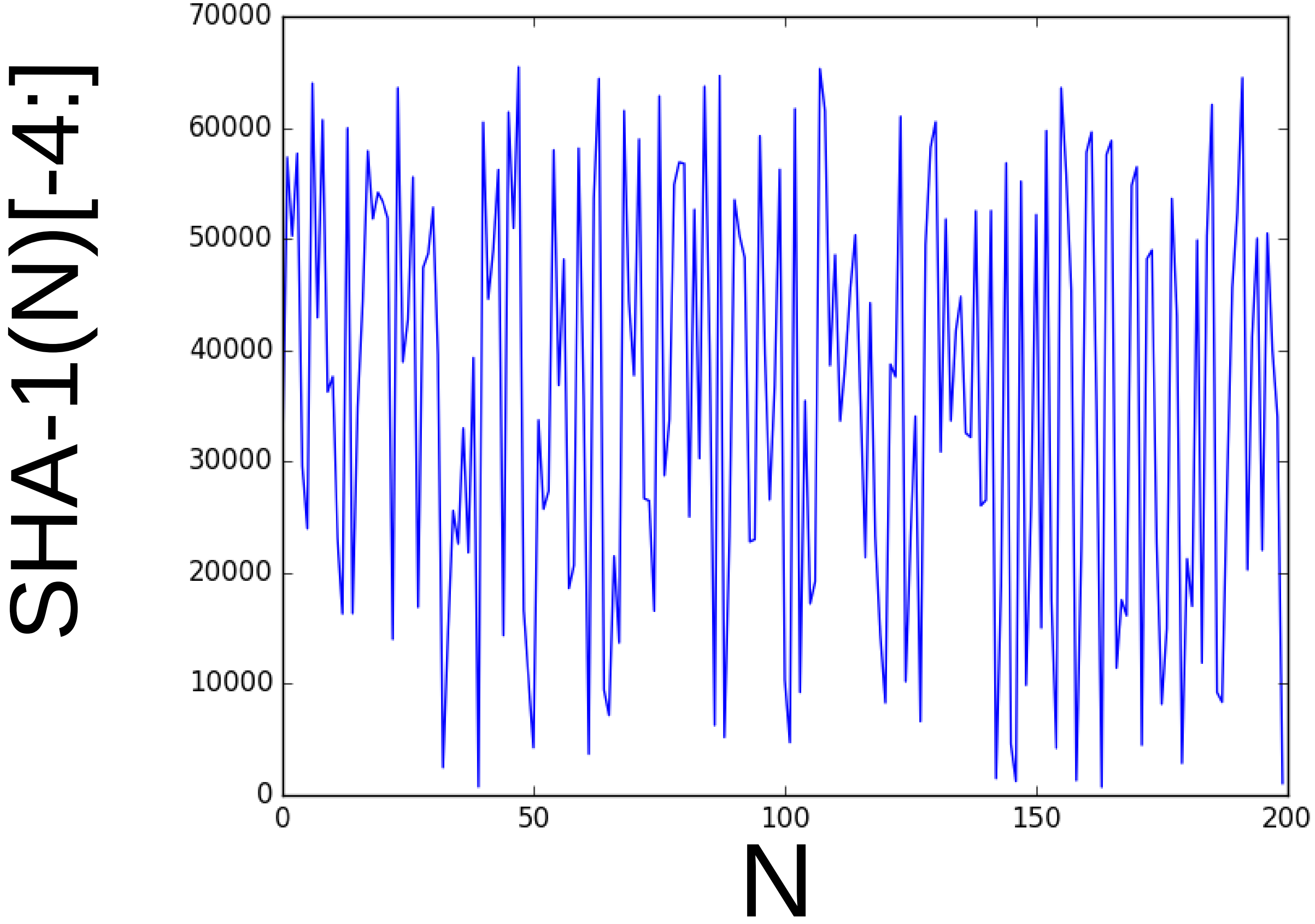
Expectation management

- Description of the attack will necessarily be general
  - This is cutting-edge cryptanalysis
  - Google *hasn’t published their code*, and the paper is vague and obtuse in places
  - There will be no demonstration :(I don’t have hundreds of GPUs or >$100K to blow on EC2

What is a hash function?

- Pseudo-random mapping of an arbitrary-length input to a fixed-length output
  - SHA-1(N) = ab3199d… (160 bits) ∀ N
- The hash of a given input is deterministic – this allows verifying identical inputs based on identical hashes
  - It is also necessarily *not* one-to-one, as a consequence of the fixed output length
- Analyzing or reversing the function should be difficult. I’ll describe specific flaws later

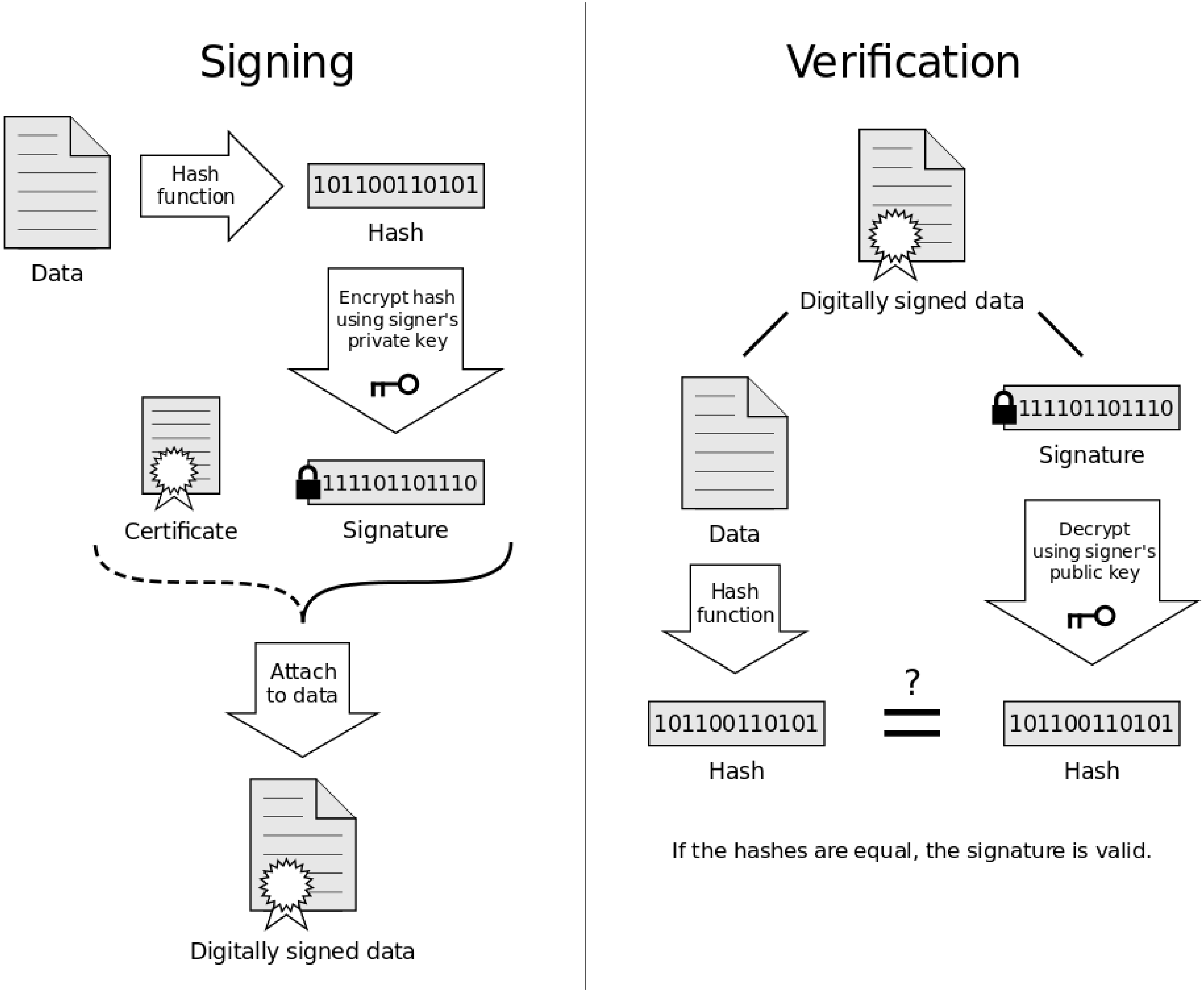
Uniform, unpredictable output



What are hashes used for?

- Verification
  - Git version control: each commit “name” is a SHA-1 hash of its contents
  - File transfers/storage: FTP, file downloads, production file systems (XFS, ZFS, Btrfs)
- Signing
  - Most signature algorithms operate only on very little data, so only a hash is signed
  - This includes TLS certificates, the basis for HTTPS

What are hashes used for?



How do hash functions fail?

- A hash function h can fail in 3 ways, ordered by decreasing severity:
  - Pre-image attack: given only a hash *h(m)*, an attacker can find a message *m* which generates that hash
  - Second pre-image attack: given a message m_1, an attacker could find a second message m_2 which generates the same hash $h(m_1) = h(m_2)$
  - Collision attack: find any two messages m_1 and m_2 for which $h(m_1) = h(m_2)$. This is the only practical attack for modern hash functions

How do hash functions fail?

- Identical-prefix attack: given identical prefixes p, attacker can find some blocks b_1, b_2 for which $h(p \,||\, b_1 \,||\, s) = h(p \,||\, b_2 \,||\, s)$
- Chosen-prefix attack: given *different* prefixes p_1, p_2, an attacker can suffixes m_1, m_2 such that

  $h(p_1 \,||\, m_1) = h(p_2 \,||\, m_2)$.
  - This is especially of interest since it allows impersonation via certificate forging, see Flame malware for an example

How practical is a Birthday Attack?

- Finding identical hashes is easier than a normal brute-force due to the birthday paradox
- SHA-1 has 160 bits of output – the work required to find a collision – any collision – is about

  $$\sqrt{\pi/2} \cdot 2^{160/2} \approx 2^{80}$$

  computations of the hash function. (This is about 10^{24})

What does SHA-1 do?

- Split input into 512-bit blocks M_1 … M_k
- Initialize a 160-bit internal state
- Operate repeatedly on the internal state, mixing in (an expansion of) each block of input via several different functions and constants

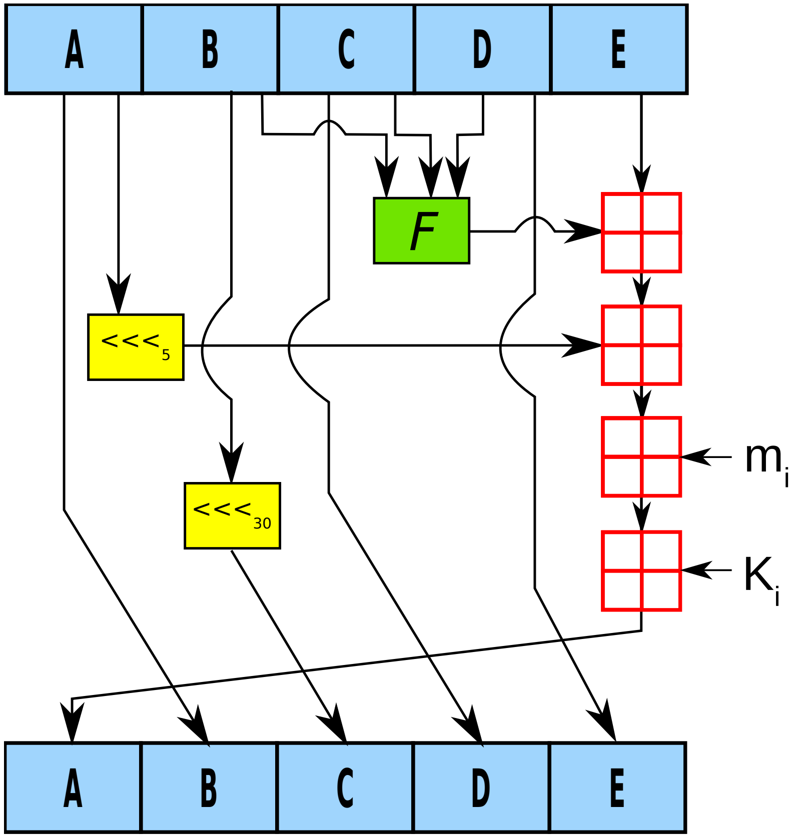
What does SHA-1 do? (Source)

```
Initialize the state
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

ml = message length in bits

Append '0' bits until length - 64 % 512 = 0
Append ml as last 64 bits

Break into 512-bit chunks.  For each:
    Break into 32-bit words $m_0$ .. $m_{15}$
    Extend those into 80 words $m_{16}$ .. $m_{79}$ via
    $m_i = (m_{i-3} \text{ xor } m_{i-8} \text{ xor } m_{i-14} \text{ xor } m_{i-16}) << 1$

Initialize the block
a,b,c,d,e = h0-4

For 80 rounds:
    Compute a function $F_i(b, c, d)$ which
    changes every 20 rounds.

    Use a constant $K_i$ which changes
    every 20 rounds

    Form a new word a by adding:
    $a = (a<<5)+F_i(b, c, d)+e+m_i+K_i$

    Shift the rest of the words
    e=d, d=c, c=(b<<30), b=a

Add the block h0 += a, h1 += b, etc.

The final hash is the concatenation of all
h0-4
```

What does SHA-1 do? (Diagram)

- Input a-e on top become output for next round on bottom
- Bitwise rotations in yellow
- Addition (mod 2^{32}) in red
- F, K change every 20 rounds

One round of SHA-1:



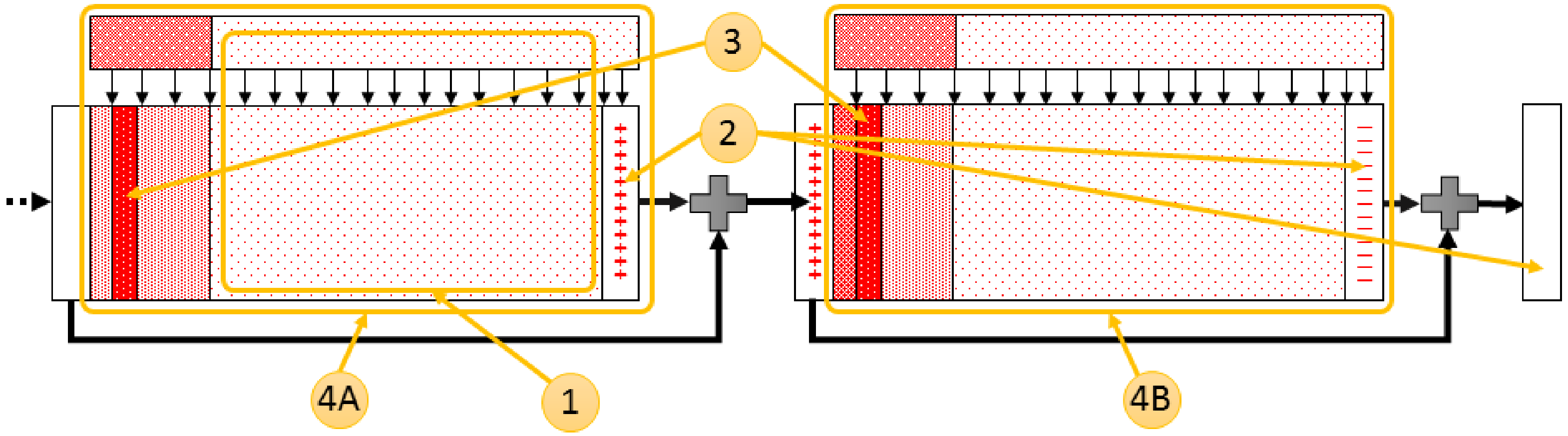
How does one attack a hash?

- SHA-1 is a streaming function: each block's result is simply added to the next
  - Thus identical prefixes and suffixes can be added at will to a set of colliding blocks
- To collide a block(s), analyze what changes to state result from a change to input
  - Find “local collisions” – differences in message bits which do not affect state within 5 rounds (remember this constitutes one rotation)
  - Then analyze “differential paths” – propagations of those disturbances through all 80 rounds of state changes

What had been done?

- There had been a lot of research into creating “good” (minimally invasive) disturbance vectors
    - Two classes of such vectors were known to the Google team, they chose a particular vector of the second class
- A good way of measuring the probability of success of a given differential path had been found
    - By the first author of the paper, Marc Stevens
    - Called “Optimal Joint-Local Collision Analysis” or JLCA

What did Google do?



- Google's attack found two blocks (4A,4B) that gave canceling contributions (2) to the internal state h0-4
- This was achieved by crafting differential paths (3) based on optimal probability of success, then computing which paths were still likely to near-collide at each step throughout the less predictable phase (1)
- These paths plus desired output resulted in a system of equations, or rather constraints. Candidates were tested against this system
- Since the first block only needed to be a near-collision, it was computed entirely on CPUs. The second was constrained to collide exactly, and so had a smaller solution space which required GPUs to guess

Disturbance Vector

- The disturbance vector is a properly expanded set $m_{0\text{-}79}$, with bits resulting in local collisions set to 1
- This provides a starting point in searching for the optimal differential path, by assuring compliance with the linear expansion that generates $m_{16\text{-}79}$
- Different disturbance vectors can be calculated based on the set of local collisions one wishes to use to construct the full near-colliding block

Differential paths

- Each run of the 80 rounds consists of
  - a “non-linear” portion: the first 16 rounds, where direct control of internal state via the input is possible
  - a “linear” portion, in which the input is derived from the message via the linear expansion function
  - These have, to my knowledge, *nothing* to do with the traditional meanings of those words
- A differential path comprises the starting state, message block, and subsequent propagation to final state
  - Thus when a desired differential path is found, it includes the desired input, in this case the colliding block

Optimal differential path

- Optimal Joint Local-Collision Analysis
  - Determines the “probability of success” of a certain path segment
  - That is, given conditions on starting state and message contents, it will produce the combination most likely to result in a collision
- Chaining together applications of the algorithm, and keeping only the most promising paths, one can construct a likely candidate for near-collision
- While determining the entire near-collision block this way would be prohibitive, it provided the first few steps' worth of internal state directly, and provided a system of equations to solve for the necessary message bits

Solving the remaining system

- Direct analysis via JLCA leaves a system of equations which can be solved to obtain the input bits
- Here, the computation of each block differs:
  - For the first block, no specific relationship had to be followed, so it was computed entirely on the CPU by trial and error
  - For the second block, a specific difference in state was required, which made the system more complicated
    - Partial solutions to step 14 were generated via JLCA on CPU, then GPUs were used to extend those solutions deterministically to step 26, and probabilistically to step 53.
    - The final candidates were then checked on CPU

Optimizations

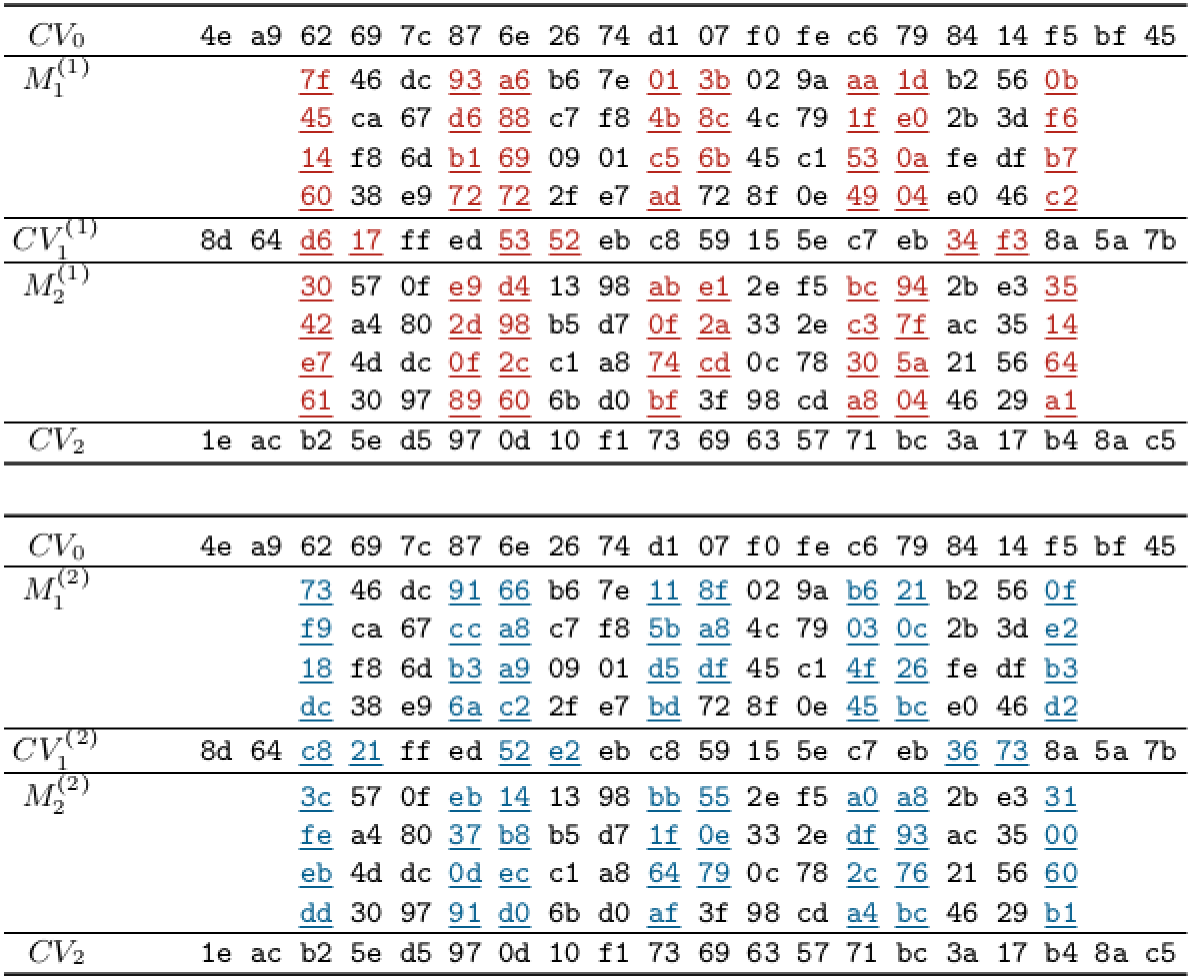
- Bits not on the differential path (to high probability), called “neutral bits” could be safely ignored until they converged with the differential path again
  - Several bits are neutral for a few steps at a time: e.g. parts c-e of state until they are rotated
- Bits which, when changed together, do not affect state for a few steps, called “boomerangs”
- These could be used to easily generate new solutions which still satisfied all requirements up to some step

Time Complexity

- Complexity was approximately the same as computing $2^{62\text{-}63}$ (or about 10^{19}) SHA-1 hashes
  - This is a pretty inaccurate, though traditional, metric, due to how different the two computational loads are
- This equated to about 3000 CPU core-years to compute the first block, and 100 GPU-years to compute the second block
- This would cost ~$100K at current Amazon EC2 spot prices

The collision

- A very scary set of numbers:



CV_0	4e	a9	62	69	7c	87	6e	26	74	d1	07	f0	fe	c6	79	84	14	f5	bf	45
$M_1^{(1)}$			7f	46	dc	93	a6	b6	7e	01	3b	02	9a	aa	1d	b2	56	0b		
			45	ca	67	d6	88	c7	f8	4b	8c	4c	79	1f	e0	2b	3d	f6		
			14	f8	6d	b1	69	09	01	c5	6b	45	c1	53	0a	fe	df	b7		
			60	38	e9	72	72	2f	e7	ad	72	8f	0e	49	04	e0	46	c2		
$CV_1^{(1)}$	8d	64	d6	17	ff	ed	53	52	eb	c8	59	15	5e	c7	eb	34	f3	8a	5a	7b
$M_2^{(1)}$			30	57	0f	e9	d4	13	98	ab	e1	2e	f5	bc	94	2b	e3	35		
			42	a4	80	2d	98	b5	d7	0f	2a	33	2e	c3	7f	ac	35	14		
			e7	4d	dc	0f	2c	c1	a8	74	cd	0c	78	30	5a	21	56	64		
			61	30	97	89	60	6b	d0	bf	3f	98	cd	a8	04	46	29	a1		
CV_2	1e	ac	b2	5e	d5	97	0d	10	f1	73	69	63	57	71	bc	3a	17	b4	8a	c5

CV_0	4e	a9	62	69	7c	87	6e	26	74	d1	07	f0	fe	c6	79	84	14	f5	bf	45
$M_1^{(2)}$			73	46	dc	91	66	b6	7e	11	8f	02	9a	b6	21	b2	56	0f		
			f9	ca	67	cc	a8	c7	f8	5b	a8	4c	79	03	0c	2b	3d	e2		
			18	f8	6d	b3	a9	09	01	d5	df	45	c1	4f	26	fe	df	b3		
			dc	38	e9	6a	c2	2f	e7	bd	72	8f	0e	45	bc	e0	46	d2		
$CV_1^{(2)}$	8d	64	c8	21	ff	ed	52	e2	eb	c8	59	15	5e	c7	eb	36	73	8a	5a	7b
$M_2^{(2)}$			3c	57	0f	eb	14	13	98	bb	55	2e	f5	a0	a8	2b	e3	31		
			fe	a4	80	37	b8	b5	d7	1f	0e	33	2e	df	93	ac	35	00		
			eb	4d	dc	0d	ec	c1	a8	64	79	0c	78	2c	76	21	56	60		
			dd	30	97	91	d0	6b	d0	af	3f	98	cd	a4	bc	46	29	b1		
CV_2	1e	ac	b2	5e	d5	97	0d	10	f1	73	69	63	57	71	bc	3a	17	b4	8a	c5

Further Reading

- Stevens, Marc, et al. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, 2017.
- Stevens, Marc. "New collision attacks on SHA-1 based on optimal joint local-collision analysis." Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer Berlin Heidelberg, 2013.
- Manuel, S. Des. Codes Cryptogr. (2011) 59: 247. doi:10.1007/s10623-010-9458-9

Extra: What are F_i and K_i ?

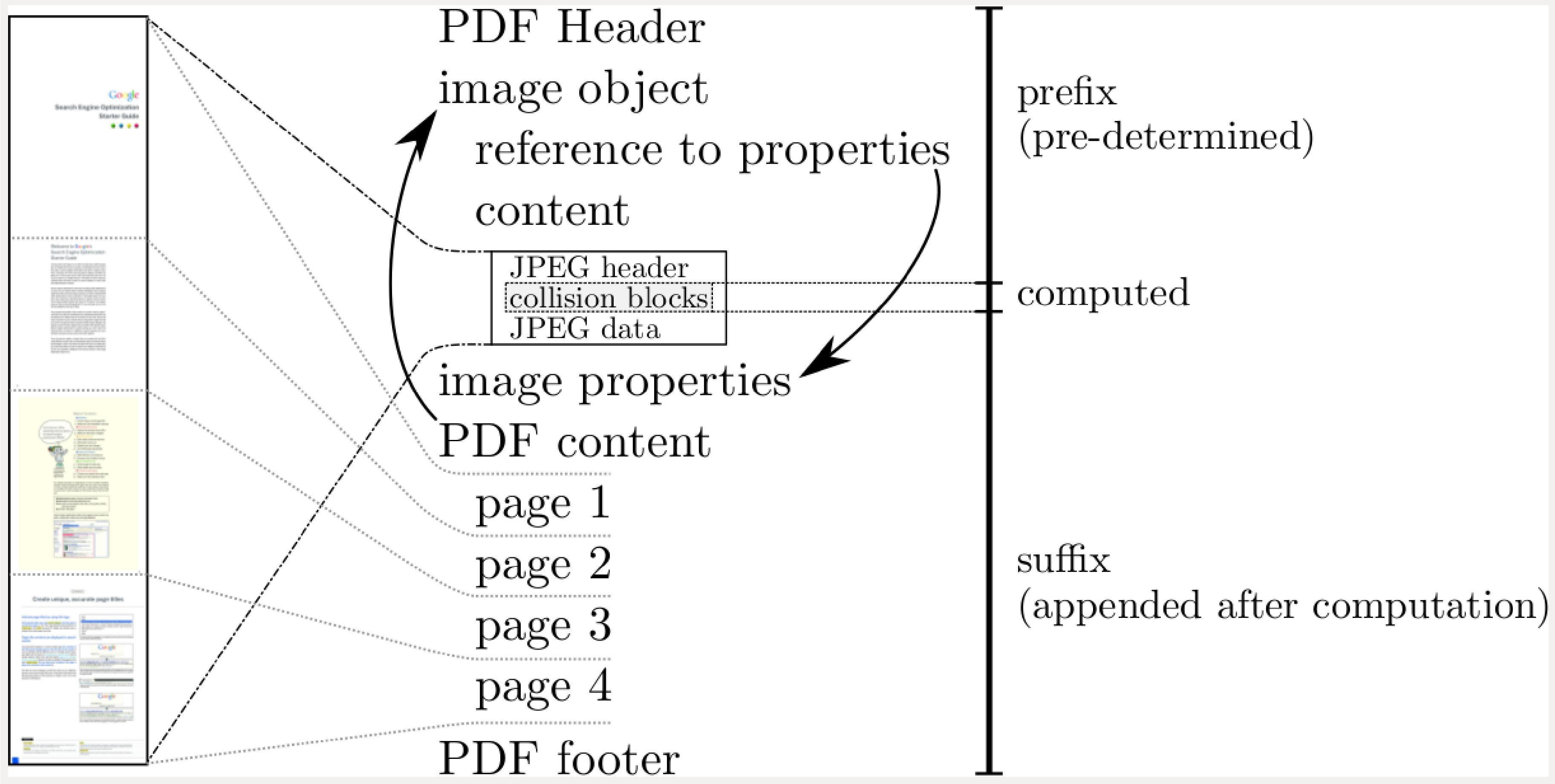
From Wikipedia's pseudocode for the inner loop:

```
for i from 0 to 79
    if 0 ≤ i ≤ 19 then
        f = (b and c) or ((not b) and d)
        k = 0x5A827999
    else if 20 ≤ i ≤ 39
        f = b xor c xor d
        k = 0x6ED9EBA1
    else if 40 ≤ i ≤ 59
        f = (b and c) or (b and d) or (c and d)
        k = 0x8F1BBCDC
    else if 60 ≤ i ≤ 79
        f = b xor c xor d
        k = 0xCA62C1D6
```

The k_i are actually just 2^{30}*sqrt(x) for x=2,3,5,10

Incidentally, the starting constants h_i are the same as those from MD5

But how did they pull the PDF trick?



PDF Header
image object
reference to properties
content
JPEG header
collision blocks
JPEG data
image properties
PDF content
page 1
page 2
page 3
page 4
PDF footer

prefix
(pre-determined)

computed

suffix
(appended after computation)