Implementation of the SHA-3 family using AVX512 instructions

Roberto Cabral, Julio López

[1]UFC-CRATEÚS - Federal University of Ceará - CE, Brazil.

[2]IC-UNICAMP - University of Campinas - SP, Brazil.

rbcabral@ufc.br, jlopez@ic.unicamp.br

Abstract. *AVX512 is the newest instruction set on the Skylake-X that extends the number of registers and provides simultaneous execution of operations over register vectors of 512 bits. This work presents how the AVX512 instruction set can be exploited to develop a fast software implementation of the Secure Hash Algorithm-3 (SHA-3) family. We achieved a speedup of around 30% when compared with x64 and AVX2 implementations. We also present a parallel implementation of two eXtendable-Output Functions (XOFs), called SHAKE128 and SHAKE256, using AVX512 that are about* $5.22\times$ *faster than a single message implementation. The SHAKE functions can be used to speedup hash-based digital signatures.*

1. Introduction

Hash functions are cryptographic primitives used in many security protocols such as digital signatures schemes, pseudo-random number generators and authentication codes. The Secure Hash Algorithms (SHA) are a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS), including: SHA-1, SHA-2 and SHA-3. The hash function SHA-1 has suffered several attacks and is no longer recommended [Wang et al. 2005], [Stevens et al. 2017]. SHA-2 is still considered secure, but its construction is based on SHA-1 and had attacks in its reduced versions, as shown in [Indesteege et al. 2009]. In 2007, NIST started a new competition to select the first cryptographic hash algorithm developed using a competition. In 2012, after 64 submissions and three rounds, the hash function Keccak [Bertoni et al. 2009] was announced as the new SHA-3 algorithm. SHA-3 is defined in [FIPS 2014] and consists of six functions SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128 and SHAKE256.

In the last few years, the processors have benefited from increasing support for vector instructions, which operate each instruction over a vector of data. The AVX512 instruction set extends the vector registers to 512 bits and the instructions can compute up to eight simultaneous 64-bit operations. In this work, we investigate how the instruction set AVX512 can be used to speed up the SHA-3 family.

The rest of the document is organized as follows: in Section 2, we describe the SHA-3 hash functions; in Section 3, is described the features of AVX512; in Section 4 we present implementation techniques for implement the SHA-3 family using AVX512 instructions; in Section 5, the performance results of our implementation are summarized; and finally, in Section 6, we present the conclusions of this work.

2. SHA-3 family

The SHA-3 family is based on the sponge construction [Bertoni et al. 2007] using the Keccak-$p[1600, 24]$ permutation; SHA-3 is specified in the NIST standard FIPS PUB 202 [FIPS 2014] and consists of six functions, four of them are cryptographic hash functions, SHA3-224, SHA3-256, SHA3-384 and SHA3-512, and two of them are *Extendable Output Functions* SHAKE128 and SHAKE256, which are cryptographic hash functions that can output an arbitrary number of bits.

The Keccak-$p[1600, 24]$ permutation has 24 rounds and works with a state of 1600 bits, which can be seen as a matrix of 5×5 of 64-bit words. The 25 words are denoted by s_i for i from 0 to 24. One round of the permutation consists of a sequence of five mappings:

- The θ mapping performs an XOR between each word of the state with the parity of the left column and the parity of the right column rotated one bit.
- The ρ mapping rotates each word a fixed number of bits.
- The π mapping permutes the words of the state.
- The χ mapping performs an XOR of each word with a non-linear function of two other words in its row.
- The ι mapping computes an XOR between word s_0 with a constant value $rc(i_r)$ in each round, where the rc values are defined in [FIPS 2014].

Given a state S and the index of a round i_r, a function `Rnd` calculates each round of the Keccak-$p[1600, 24]$ permutation as follow:

$$\texttt{Rnd}(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r),$$

and a call to the Keccak-$p[1600, 24]$ permutation consists of 24 interactions of `Rnd`.

3. Intel Micro-Architecture

The Single Instruction Multiple Data (SIMD) is an interesting trend of micro-architecture where processors contain a particular bank of vector registers and vector instructions, which can perform the same instruction on every element stored in the vector register.

In 1997 was launched one of the first set of instructions to implement the SIMD paradigm; known as MMX, added 64-bit registers and vector instructions that enabled the processing of two 32-bit operations simultaneously [Corporation 2008]; this instruction set was followed by Streaming SIMD Extensions (SSE) [Corporation 2009] that extended the size of vector registers to 128 bits. In 2011, the Advanced Vector eXtensions (AVX) instruction set extended the size of the vector registers to 256 bits and in 2013 Intel released the AVX2 instruction set [Corporation 2011], which support integer arithmetic and new instructions like variables shifts, which are useful for cryptographic algorithms.

In 2017 Intel released the Skylake-X micro-architecture with the AVX512 instruction set [Cooperation 2016], this micro-architecture doubles the number of vector registers from 16 in AVX2 to 32 in AVX512. In the following, we detail the most relevant AVX512 instructions used in this work, which will be referred by a mnemonic described in Table 1:

- **Logic**. The logic operations were extended to operate over every bit of 512-bit registers. The `TERNARY` instruction allows to logically implement all possible bitwise operations between three inputs; it takes three registers as input and an 8-bit value $imm8$. Each bit in the output is generated using a lookup of the three corresponding bits in the inputs to select one of the 8 positions in the $imm8$.
- **Rotation**. Until now the Intel SIMD instruction sets did not include a rotation instruction; with the `ROT` instruction, we can rotate n bits in each packed 64-bit integer. There is also the `ROTV` instruction that allows us to rotate the bits in each packed 64-bit integer by a specified value.
- **Combination**. The `BLEND` instruction fills the content of a vector register with the words from two different register sources chosen through a binary selection constant. The `PRBLEND` instruction combines words from two register sources chosen through a binary selection mask register.
- **Permutation**. The `PERM` instruction moves the words stored in a vector register using a permutation pattern that can be specified by a binary selection mask register.

Table 1 shows some of the most relevant instructions used in this work. In the first column is showed the type of the instructions; in the second column a mnemonic is used for each vector instruction; in the third column is described the specific assembler name of the vector instruction, and the last columns are shown the latency and the reciprocal throughput of every vector instruction; the entries were taken from Agner Fog's measurements published in [Fog 2018a].

Table 1. Latency and reciprocal throughput of some AVX512 instructions.

Type	Mnemonic	Assembler Instructions	Latency (cycles)	Reciprocal Throughput (cycles/op)
Logic	TERNARY	`vpternlogq`	1	0,5
	AND/XOR	`vpandd/vpxord`	1	0.5
Rotate	ROT	`vprolq/vprorq`	1	1
	ROTV	`vprolvq`	1	1
Combination	BLEND	`vpblendmq`	1-2	0.5
	PRBLEND	`vpermi2q`	3	1
	UNPACK	`vpunpckhqdq/vpunpcklqdq`	1	1
Permutation	PERM	`vpermq`	3	1

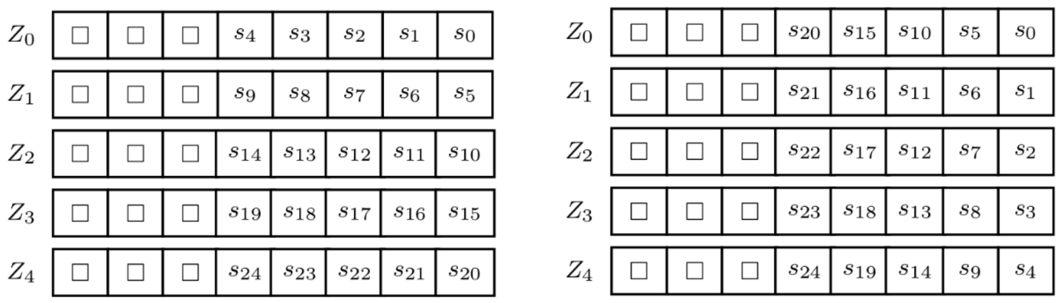
In Skylake-X micro-architecture there are two 256-bit vector execution units at ports 0 and 1; these units can be combined into one 512-bit unit to execute instructions of 512 bits; there is an additional 512-bit vector unit under port 5, and the permute instructions and other instructions that may move byte across the 128-bit lane boundaries are always handled by port 5 [Fog 2018b]. Some common integer vector instructions (up to 256 bits) have a throughput of three instructions per clock using ports 0,1 and 5, while the 512-bit versions of the same instruction have a throughput of two instructions per clock, going through ports 0 and 5.

4. Implementations

The Keccak-$p[1600, 24]$ permutation is the same for all the flavors of the SHA-3 family and is the main responsible for the efficiency. In this section, we will show how the new instruction set AVX512 can be used to speed up the Keccak-$p[1600, 24]$ permutation; this function has 24 rounds and uses a state of 25 words of 64 bits $S = [s_0, \ldots, s_{24}]$.

4.1. Sequential version using 512-bit registers

The 512-bit registers allow us to gather until eight 64-bit words in a single vector register; aiming a better use of the instruction set, we use a vector of five 512-bit registers $[Z_0, \ldots, Z_4]$ to represent the matrix state into register vectors, as can be see in Figure 1(a); this organization reduces the number of permutation instructions resulting in a fast and compact code.



Z_0	$\square$	$\square$	$\square$	s_4	s_3	s_2	s_1	s_0
Z_1	$\square$	$\square$	$\square$	s_9	s_8	s_7	s_6	s_5
Z_2	$\square$	$\square$	$\square$	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}
Z_3	$\square$	$\square$	$\square$	s_{19}	s_{18}	s_{17}	s_{16}	s_{15}
Z_4	$\square$	$\square$	$\square$	s_{24}	s_{23}	s_{22}	s_{21}	s_{20}

(a) State organization using 512-bit registers.

Z_0	$\square$	$\square$	$\square$	s_{20}	s_{15}	s_{10}	s_5	s_0
Z_1	$\square$	$\square$	$\square$	s_{21}	s_{16}	s_{11}	s_6	s_1
Z_2	$\square$	$\square$	$\square$	s_{22}	s_{17}	s_{12}	s_7	s_2
Z_3	$\square$	$\square$	$\square$	s_{23}	s_{18}	s_{13}	s_8	s_3
Z_4	$\square$	$\square$	$\square$	s_{24}	s_{19}	s_{14}	s_9	s_4

(b) State organization after performing the π step.

Figure 1. State organization.

The θ mapping performs an XOR between each word of the state with the parity of the left column and the parity of the right column rotated one bit. To compute the parity of the columns required in the θ mapping, we use two `TERNARY` instructions to compute an XOR among Z_0, Z_1, Z_2, Z_3 and Z_4, resulting C=$[\square, \square, \square, c_4, c_3, c_2, c_1, c_0]$ [1]. Thereafter, we use two permutations to calculate C_1=$[\square, \square, \square, c_3, c_2, c_1, c_0, c_4]$, C_2=$[\square, \square, \square, c_0, c_4, c_3, c_2, c_1]$ and we perform C_1=`ROT` (C_1,0x01) to rotate one bit in each word of C_1. To update the vector state after the θ mapping, it is performed $Z_i = \texttt{TERNARY}(Z_i, C_1, C_2, \texttt{XOR})$ for i from 0 to 4.

The mappings ρ and π consist in the application of a rotation and a permutation on the words of the state, respectively. The `ROTV` instruction allows us to implement the ρ mapping efficiently by using only five instructions. After performing the π mapping we organize the state as shown in Figure 1(b), which allows us to perform the χ mapping as $Z_i = \texttt{TERNARY}(Z_i, Z_{[(i+1 \bmod 4)]}, Z_{[(i+2 \bmod 4)]}, \texttt{XORnotAND})$ for i from 0 to 4.

Towards the next rounds, we need to reorganize the state to the initial configuration; Algorithm 1 shows an implementation based on AVX512 instructions to reorganize the state. To complete a round, the ι mapping performs an XOR between the word s_0 and the constant values $rc(i_r)$.

4.2. Parallel version using 512-bit registers

The AVX512 instruction set allows us to see each 512-bit vector register as eight words of 64 bits and compute an operation on every element stored in the vector register; this feature allows us to group eight 64-bit states $S^0 = [s_0^0, \ldots, s_{24}^0]$, ..., $S^7 = [s_0^7, \ldots, s_{24}^7]$ into state represented by $[Z_0, Z_1, \ldots, Z_{24}]$, where the first words from each 64-bit state

[1]The c_i values are the parity of each line i and $\square$ represents unused values in the register.

Algorithm 1 State organization for initial configuration

```
1: $T_1$ = PRBLEND ($Z_0$,$P_1$,$Z_1$);
2: $T_2$ = PRBLEND ($Z_2$,$P_2$,$Z_3$);
3: $T_3$ = BLEND (0xCC,$T_1$,$T_2$);
4: $T_4$ = BLEND (0x33,$T_1$,$T_2$);
5: $T_5$ = PRBLEND ($Z_1$,$P_3$,$Z_3$);
6: $Z_0$ = PRBLEND ($T_3$,$P_4$,$Z_4$);
7: $Z_1$ = PRBLEND ($T_4$,$P_5$,$Z_4$);
8: $Z_2$ = PRBLEND ($T_3$,$P_6$,$Z_4$);
9: $T_2$ = BLEND (0x10,$T_4$,$Z_4$);
10: $T_3$ = BLEND (0x0F,$T_4$,$T_5$);
11: $Z_3$ = PRBLEND ($T_3$,$P_7$,$Z_4$);
12: $Z_4$ = PRBLEND ($T_2$,$P_8$,$T_5$);
```

will be stored into the first register Z_0, the second words in the following register Z_1 and so on. This rearrangement allows us to reduce the permutations overhead.

In order to compute the hash of eight messages concurrently, the first step is to map the eight x64-bit states into a 512-bit state. Algorithm 2 presents the mapping to the first eight registers, the other words are mapped similarly.

Algorithm 2 Parallel state organization

```
Input: Eight x64-bit states, $S^0, \ldots, S^7$ .
Output: 512-bit state $Z_0, \ldots, Z_{24}$.
1: for i=0; i<8; i++ do
2:     $T_i$ = LOAD($S^i$,0)
3: end for
4: for i=0; i<8; i=i+2 do
5:     $R_i$ = UNPACK($T_i$,$T_{i+1}$)
6:     $R_{i+1}$ = UNPACK($T_i$,$T_{i+1}$)
7: end for
8: for i=0; i<8; i=i+4 do
9:     $T_i$ = PRBLEND($R_i$,$m_0$,$R_{i+2}$)
10:    $T_{i+1}$ = PRBLEND($R_i$,$m_1$,$R_{i+2}$)
11:    $T_{i+2}$ = PRBLEND($R_{i+1}$,$m_0$,$R_{i+3}$)
12:    $T_{i+3}$ = PRBLEND($R_{i+1}$,$m_1$,$R_{i+3}$)
13: end for
14: for i=0; i<2; i++ do
15:    $Z_{i*2}$ = PRBLEND($T_i$,$m_2$,$T_{i+4}$)
16:    $Z_{i*2+1}$ = PRBLEND($T_{i+2}$,$m_2$,$T_{i+6}$)
17:    $Z_{i*2+2}$ = PRBLEND($T_i$,$m_3$,$T_{i+4}$)
18:    $Z_{i*2+3}$ = PRBLEND($T_{i+2}$,$m_3$,$T_{i+6}$)
19: end for
```

After mapping the state to 512-bit registers, we start the computation of the Keccak-$p[1600, 24]$ permutation; this implementation uses 512-bit instructions to process eight independent states concurrently. The new instruction `TERNARY` allows us to reduce by half the number of XORs in the θ mapping. Other crucial instruction is the `ROT` that processes one rotation with a single instruction instead of two shifts and one XOR.

For the efficient use of vector instructions and registers, one can perform the mappings ρ, π and χ modularly, by processing a block of five registers at a time. A block needed to compute a row in the χ mapping is performed by the ρ and π mappings. In the ρ mapping is used the `ROT` instruction to rotate a fixed amount of bits in each word of the state. The π mapping permutes the words of the state and its implementation does not require additional instructions, only renaming registers; for instance, the words of $[s^0_{12}, s^1_{12}, \ldots, s^7_{12}]$ stored into the register Z_{12} will become the words of $[s^0_2, s^1_2, \ldots, s^7_2]$ after performing π.

The χ mapping uses the `TERNARY` instruction to performs an XOR of each word with a non-linear function of two other words in its row. Algorithm 3 shows the computation of the ρ, π and χ mappings for the first block, the other blocks are performed similarly.

For a better use of the 512-bit registers, instead of rearranging the words after

Algorithm 3 The ρ, π and χ steps for one block

```
1: $T_0 =$ ROT($Z_0$,0x00)
2: $T_1 =$ ROT($Z_6$,0x2C)
3: $T_2 =$ ROT($Z_{12}$,0x2B)
4: $T_3 =$ ROT($Z_{18}$,0x15)
5: $T_4 =$ ROT($Z_{24}$,0x0E)
6: $Z_0 =$ TERNARY($T_0$,$T_1$,$T_2$, XORnotAND)
7: $Z_{12} =$ TERNARY($T_1$,$T_2$,$T_3$, XORnotAND)
8: $Z_{24} =$ TERNARY($T_2$,$T_3$,$T_4$, XORnotAND)
9: $Z_6 =$ TERNARY($T_3$,$T_4$,$T_0$, XORnotAND)
10: $Z_{18} =$ TERNARY($T_4$,$T_0$,$T_1$, XORnotAND)
```

these steps, we compute the next round taking into a count the new organization of the state and returning to the original configuration only in the following round.

The versions that process two and four states concurrently using AVX512 are very similar to this one, changing only the state organization and the registers. These implementations without AVX512 tend to be slower because there is not support in AVX2 (and previous instruction sets) for the `TERNARY` and `ROT` instructions.

5. Performance Results

Benchmarking was performed on a Skylake-X processor (Core i7-7820X) at 3.6 GHz, where the Intel Turbo Boost and Intel Hyper Threading technologies were disabled to ensure the reproduction of experiments. Our source code[2] was performed on Fedora 27 using the clang compiler version (5.0.2) and the timings were measured as the average time of 10^4 computations.

Table 2 shows the performance results for messages of 4096 bytes, in cycles per byte, for different implementations of the SHA-3 family; we selected 1344 and 1088 bits as the digest size of SHAKE128 and SHAKE256, respectively. The first column exhibits our AVX512 implementations presented in Section 3, the others columns show the implementations available in eBASH [Bernstein and Lange 2018] optimized to AVX512, AVX2 and x64, respectively, all the timings were measured in our machine.

The x64-supercop native implementation is faster than the version using AVX2; this is because the AVX2 implementation needs a lot of slow permutation instructions. For the x64 implementation, on other hand, the permutation instructions are just referencing changes in memory. Our AVX512 implementation improves the performance of the SHA-3 family; this improvement was mainly due to the representation of the state as a vector of five registers of 512 bits, which reduced the number of permutation instructions.

Some applications, such as the Merkle Digital Signature [Merkle 1990], require the computations of the hash values of several messages with the same length. The new RFC XMSS: eXtended Merkle Signature Scheme [Huelsing et al. 2018] and the NIST Post-Quantum Cryptography candidate SPHINCS+ [Bernstein et al. 2018] are examples of hash-based digital signature schemes that need multiple calls to SHAKE256. In this context, Table 3 shows the performance and speedup for three parallel implementations of the SHAKE128 and SHAKE256 functions to process 4096 bytes. The 1-way implementation is the sequential version, the 2-way uses 128-bit registers to process two messages, the 4-way uses 256 bits to process four messages, and the 8-way uses 512-bit registers to process eight messages concurrently.

[2]Our source is available in `https://github.com/rbCabral/SHA-3`.

Table 2. Cycles per byte of the SHA-3 Family on Skylake-X.

	Our work	AVX512-supercop	AVX2-supercop	x64-supercop
SHA3-224	6.02	6.41	8.42	8.28
SHA3-256	6.44	6.80	9.00	8.84
SHA3-384	8.30	8.86	11.59	11.33
SHA3-512	11.84	12.46	16.46	16.00
SHAKE128	5.21	5.27	7.34	7.28
SHAKE256	6.43	6.79	8.99	8.8

	SHAKE128		SHAKE256	
	Cycles per bytes	Speedup	Cycles per bytes	Speedup
1-way (512-bit register)	5.21	1.00	6.47	1.00
2-way (128-bit register)	2.86	1.82	3.48	1.86
4-way (256-bit register)	1.48	3.52	1.82	3.55
8-way (512-bit register)	1.03	5.06	1.24	5.22

Table 3. Cycles per bytes and speedup of parallel implementations.

As can be seen in Table 3, the AVX512 instruction set allows a significant improvement for the parallel performance of SHAKE; this is due to a duplication on the number of vector registers, from 16 to 32, and the release of some new instructions, such as `TERNARY` and `ROT`.

The speedup from the 2-way to the 4-way SHAKE256 implementation is 1.90 while the speedup from the 4-way to the 8-way is just 1.47; this happens because as discussed in Section 3 the Skylake-X micro-architecture has two execution ports for instructions that use 512-bit registers while having three execution ports for instructions over 128 and 256 registers.

6. Conclusions

In this work we present a fast software implementation of the SHA-3 family of cryptographic hash functions using the new vector instruction set AVX512. The use of AVX512 brings a speedup of around 30% when compared with native (x64) and AVX2 implementations; this gain comes from the increase in size of the vector registers that allow us to organize the state as a vector of five registers of 512 bits, reducing the number of permutation instructions in the implementation. AVX512 also brings the `ROTV` and `TERNARY` instructions which reducing significantly the number of instructions needs for the implementation of the ρ and χ mappings.

AVX512 was used to produce eight hash values concurrently with a speedup of $5.22\times$, this 8-way implementation can be used to speedup hash-based digital signature schemes, such as XMSS and SPHINCS+. We noted that if the performance of the 8-way implementation remained like the 2-way and 4-way, we should have a speedup at around $7.2\times$; this speedup was not achieved because the Skylake-X micro-architecture has only two execution ports for instructions that use 512-bit registers and three execution ports for instructions over 128 and 256 vector registers.

References

Bernstein, D., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S. L., Huelsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., and Schwabe, P. (2018). Sphincs+: Submission to the nist post-quantum project. Technical report.

Bernstein, D. J. and Lange, T. (2018). ebacs: Ecrypt benchmarking of cryptographic systems.

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. (2009). Keccak specifications. *Version 2 (as updated for round 2)*.

Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. (2007). Sponge functions. In *ECRYPT hash workshop*, volume 2007.

Cooperation, I. (2016). Intel architecture instruction set extensions programming reference. *Intel Corp., Mountain View, CA, USA, Tech. Rep*, pages 319433–030.

Corporation, I. (2008). Intel® Pentium processor with MMX$^{\text{TM}}$ technology documentation. `http://www.intel.com/design/archives/Processors/mmx/`.

Corporation, I. (2009). Define SSE2, SSE3 and SSE4. `http://www.intel.com/support/processors/sb/CS-030123.htm`.

Corporation, I. (2011). Intel® Advanced Vector Extensions Programming Reference. `https://software.intel.com/sites/default/files/m/f/7/c/36945`.

FIPS, P. (2014). Secure hash algorithm-3 (sha-3) standard: Permutation-based hash and extendable-output functions. *National Institute for Standards and Technology (NIST)*, 202(0).

Fog, A. (2018a). *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*.

Fog, A. (2018b). *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*.

Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and Mohaisen, A. (2018). Xmss: extended merkle signature scheme. Technical report.

Indesteege, S., Mendel, F., Preneel, B., and Rechberger, C. (2009). Collisions and other non-random properties for step-reduced sha-256. In *Selected Areas in Cryptography*, pages 276–293. Springer.

Merkle, R. (1990). A certified digital signature. In Brassard, G., editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer New York.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer.

Wang, X., Yin, Y. L., and Yu, H. (2005). Finding collisions in the full sha-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer.