Efficient Hash Collision Search Strategies on Special-Purpose Hardware

Tim Güneysu, Christof Paar, Sven Schäge

Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{gueneysu,cpaar}@crypto.rub.de, sven.schaege@nds.rub.de

Abstract

Hash functions play an important role in various cryptographic applications. Modern cryptography relies on a few but supposedly well analyzed hash functions which are mostly members of the so-called MD4-family. This work shows whether it is possible, using special-purpose hardware, to significantly speedup collision search for MD4-family hash functions. A thorough analysis of the computational requirements for MD4-family hash functions and corresponding collision attacks reveals that a microprocessor based architecture is best suited for the implementation of collision search algorithms. Consequently, we designed and implemented a (concerning MD4-family hash-functions) general-purpose microprocessor with minimal area requirements and, based on this, a full collision search unit. Comparing the performance characteristics of both ASICs with standard PC processors and clusters, it turns out that our design, massively parallelized, is nearly four times more cost-efficient than parallelized standard PCs. With further optimizations, we believe that this factor can even be improved.

Keywords. Hash functions, Special-purpose Hardware, Crypto Attacks.

1 Introduction

Many basic and complex cryptographic applications make extensive use of cryptographic hash functions. They offer valuable security properties and computational efficiency. In combination, these features are particularly interesting for accelerating asymmetric cryptographic protocols. Usually, the security of a cryptographic protocol is dependent on all its elements. If just one primitive can be found with security flaws, the whole protocol might become insecure. Finding successful attacks against widespread cryptographic hash functions would affect a variety of popular security protocols and have unforeseeable impact on their overall security [10, 11].
In February 2005 Wang et al. presented a new attack method against the popular Secure Hash Algorithm (SHA-1). It reduces the computational attack complexity to find a collision from 2^{80} to approximately 2^{69} compression function evaluations [20] leading to the announcement that SHA-1 has been broken in theory. In 2006, it was further improved to about 2^{62} compression function calls [22]. However, still this attack is supposed to be theoretic in nature, because the necessary number of computations is very high.
For practical attacks, all theoretical results have to be mapped to an executable algorithm, which subsequently has to be launched on an appropriate architecture. Basically, there are two ways to design such architectures, namely standard and special-purpose hardware.
Generally, both FPGA and ASIC architectures require higher development costs than PC based systems. However, at high volumes special-purpose hardware is usually superior to PC clusters with respect to cost-efficiency.
The main issue of this work is whether it is possible to develop alternative hardware architectures for collision search which offer better efficiency than the aforementioned standard PC architectures. Given a certain amount of money, which hardware architecture should be invested in to gain best performance results for collision search and make practical attacks feasible?
Our solution is a highly specialized, minimal ASIC microprocessor architecture called μMD.

μMD computes 32-bit words at a frequency of about 303 MHz. It supports a very small instruction set of not more than 16 instructions and, in total, needs an area of just 0.022341 μm^2. For collision search, μMD is connected via a 32-bit bidirectional bus to an on-chip memory and I/O module, resulting in a standalone collision search unit called μCS.
In literature, there are a few publications that deal with practical issues of collision search algorithms [3, 4]. Most of the work is dedicated to rather theoretical problems. Current implementations of MD5 collision search algorithms for PC systems are given in [7, 9, 19]. Joŝĉák [7] compares their performances in detail. However, other types of architectures are not considered. To our best knowledge, this is the first work that analyzes implementation requirements for collision search algorithms from an algorithmic and architectural perspective.
This work is structured as follows. Section 2 gives an introduction to the basic features of MD4-family hash functions. In contrast to this, Section 3 gives an overview over current attacking techniques on MD4-family hash functions. In Section 4, we derive from these techniques concrete design and implementation requirements for our target architecture. Subsequently, in Section 5, we give a detailed description of our final collision search architecture. In Section 6, we then develop a metric to adequately compare the performances of different hardware circuits for collision search. This metric is then applied to our final architecture, providing detailed information on its performance. We close with a short conclusion.

2 Hash Functions of the MD4-family

A (cryptographic) hash function is an efficiently evaluable mapping h which maps arbitrary-sized messages to fixed-size hash values [6, 13].

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

There are at least three features a secure hash function is expected to have; (first) preimage resistance, second preimage resistance, and collision resistance. Successful attacks on collision resistance, i.e. finding two distinct messages that map to the same hash value, are much more promising, hence most attacks in the literature focus hereon.
It is hard to efficiently describe algorithms that directly process inputs of variable length. To cope with variable input length, hash functions of the MD4-family pad the input message M and divide it into fixed-size message blocks

$$M = M_1|M_2|\ldots|M_{q-1},$$

where each block M_i for $i = 1, \ldots, q-1$ is r bits long. These blocks are then successively processed by a so called compression function f.

$$f : \{0,1\}^n \times \{0,1\}^r \rightarrow \{0,1\}^n$$

To impose chaining dependencies between successive message blocks the compression function evaluation also processes the output of the preceding computation, which is in this context also called chaining value cv. The first chaining value is a fixed initialization vector IV. The output of the computation of the last message block is defined to be the output of the entire hash function. Hash functions of the MD4-family differ mainly in the design of their compression function and their initialization vector.

$$\begin{aligned} cv_0 &= IV \\ cv_{i+1} &= f(cv_i, M_i),\ 1 \leq i \leq q-1 \\ h(M) &= cv_q \end{aligned}$$

Hash functions of the MD4-family are constructed in line with the Merkle-Damgård Theorem [14, 15]. This theorem delivers a useful security reduction stating that the security of a hash function can be concluded from the security of its compression function: if it is computationally infeasible to find two distinct pairs of inputs to a compression function that map to the same output (*pseudo collision*), then it is hard to find a collision of the hash function. Therefore, most attacks on hash functions of the MD4-family actually target compression functions. However, there is no general way known to exploit a single (random) pseudo collisions.
Practically, there are two ways to find useful colliding messages differing in the number of message blocks required to generate a collision. Single block collisions generate a collision using a single pair of distinct message blocks. Contrarily, multi-block collisions use several pairs of messages that, successively computed in a predefined order, result in a colliding output. Usually, the intermediate outputs of the compression function only differ in very few bit positions. Such situations are referred to as near-collisions [1].

3 Attacks on MD4-family Hash Functions

Generally, there are two types of attacks on MD4-family hash functions, generic and specific attacks.
Generic attacks are attacks that are applicable to all (even ideal) hash functions. Using a generic attack for finding a collision for a hash function with output size n requires computational complexity of $O(2^{\frac{n}{2}})$. This result is due to the so-called birthday attack [23]. The birthday attack basically exploits a probabilistic result that is commonly known as the birthday paradox or the birthday collision.
Specific attacks try to exploit the knowledge of the inner structure of the hash function and its inherent weaknesses. In this way, it is possible to *construct* collisions to a certain extent. Specific attacks are always dedicated to a single hash function. However, there are some general methods to develop such attacks on MD4-family hash functions.
At present, successful attacks can be divided into two phases. The first phase launches a differential attack [2] on the inner structure of the compression function. Essentially, this method exploits the fact that collisions can adequately be described using differences. A collision is just a pair of messages with a difference unequal to zero that maps to a zero output difference. Differences may propagate through parts of the compression function in a predictable way. Therefore, the goal is to find conditions under which useful differences propagate with a high probability. All these identified conditions, the so-called differential path, are then mapped into a search algorithm. Assuming a random traversal over the differential path, the number of conditions reflects the search complexity of the algorithm.
The second phase of a specific attack consists of utilizing the remaining freedom of choice for the message bits. Of course, this freedom can be used to predefine parts of the input messages. However, another and very popular application is to exploit it for a *significant* acceleration of the collision search. In the literature one can find single-step modifications [21], multi-step modifications [8, 12, 18, 21], and tunneling [9]. Roughly speaking, these techniques consist of determining bits in the computation path such that, if altered in an appropriate way, they do not influence preceding, yet fulfilled conditions. Randomly choosing new message pairs and checking if all conditions up to this position are satisfied can mean high computational costs and have a high failure probability. Instead, by exploiting these methods, it is possible to deduce new message pairs with the same characteristic based on a single pair of messages satisfying all conditions so far. Usually this deduction is computationally much cheaper and less probable to fail. Advantageously, the number of new messages increases exponentially with the number of found bits. This provides for a significant increase in efficiency.

4 Architecture Requirements

When analyzing MD4-family hash functions and their corresponding collision search algorithms, one can find several important hints how a suitable hardware architecture should be designed. The computation process starts by randomly choosing two messages with a fixed difference, what practically rises the need for a pseudo random number generator. Then, it applies computations of the compression function (*step iterations*) to the state variables. Since hash functions of the MD4-family have been developed also with respect to PC based architectures, all step computations require similar operation sets. By implementing a few boolean, arithmetic and bit rotation operations, all requirements of the entire MD4-family can be met.
However, the collision search algorithms require additional arithmetic and flow operations. Instead of choosing new messages and recomputing all steps so far, the aforementioned acceleration techniques rather adapt new messages to fulfill conditions. Therefore, step equations have to be rearranged and solved for the message bits. For MD4-family hash functions, such rearrangements are not difficult. However, they require additional operations like the subtraction operator, which is usually not used in the original hash function specifications.
Subsequent to many computations, single data dependent bits are compared with conditions of the differential path. When satisfied, the algorithm proceeds, otherwise it returns to an earlier position in the computation path. Computationally, this requires (conditional) branches.
Hash functions of the MD4-family have been developed for fast software execution on standard PCs [17]. They operate on data units with popular processor word lengths 32-bit or 64-bit, and all their operations consist of typical processor instructions. As a consequence, the target hardware for the collision search algorithm should also work on 32-bit or 64-bit words. There are frequent operations in the hash function, like modular additions and bit rotations that do not only operate on a single pair of bits but propagate changes among neighbouring bits as well. In contrast to bit-wise defined operations, like AND, OR, NOT, the actual effect of such operations heavily depends on the processor word length. When operands are divided up into subparts (e.g., chunks of 8 bits), the original impact across several bit positions may require additional processing. To guarantee compliance with the specified hash algorithm (or its collision search algorithm) these results have to be corrected in a post processing step.
Collision search algorithms can hardly be parallelized on lower hierarchical levels due to their strictly sequential structure. Useful situations are confined to operations within the step function where the evaluation of two modular additions can be computed concurrently. In almost all other cases this is not possible, since most operations also process the result of their immediate predecessor.
Besides multi step modifications, we believe that tunneling will become a standard tool for improving collision search based on differential patterns. Unfortunately, tunneling highly parameterizes the computation path using loop constructions. This requires efficient resource reuse. In combination with frequent instruction branching, this fact renders hardware acceleration techniques like pipelining hardly useful.
For MD5 [17], there exist several efficient collision search algorithms [7, 9, 19]. Joŝčák [7] compares their performances in more detail, where Klima's collision search (CS) approach [9] turns out to be the fastest. In contrast to the other ones, this algorithm extensively makes use of tunneling. CS is divided into two parts, reflecting the structure of a multi collision. The first part searches for a near-collision of the compression function, given the standard initialization vector. The second part generates an appropriate pseudo collision.

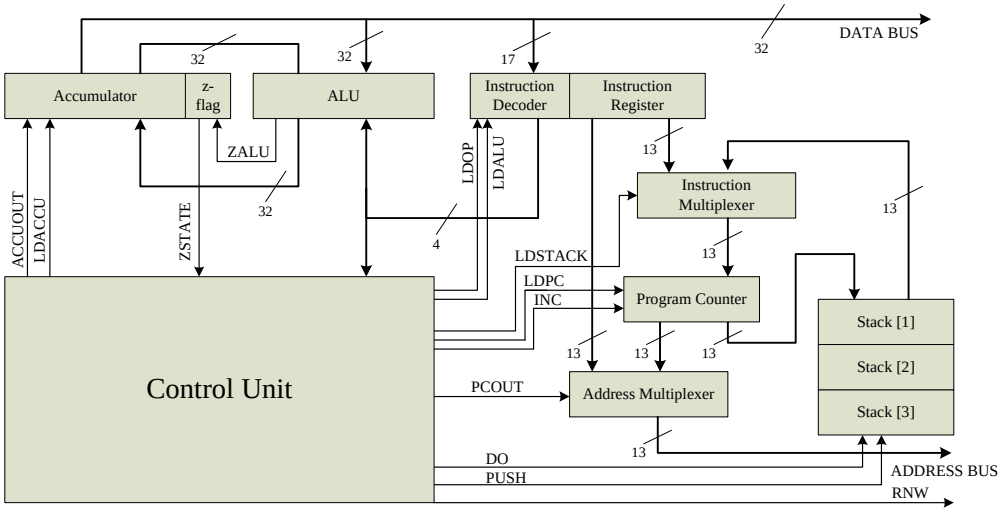

Figure 1: Simplified architecture of collision search processor μMD

5 Architecture Design

5.1 Design Process

The development process can be divided into several phases. In the first step we designed the basic processor μMD using a VHDL integrated development environment.
For a verification and performance analysis, we additionally required appropriate memory devices containing the program code and constants and offering enough space for storing temporary variables. We denote the assembly of those components with μCS.
For simulation purposes, we developed tools to automatically load the future ROM modules of μCS with the content of dedicated binary files. To generate these files, we had to develop a dedicated assembly language. In the next step we had to program ACS, the assembly version of CS, to gather information on the required memory space of μMD and determine some basic facts about its runtime behavior. Unfortunately, it is currently not possible to thoroughly analyze ACS long-run behavior in the simulation model for gaining average values. It is even hardly possible to observe a single collision, when the algorithm is once started.

5.2 Microprocessor Design

For the given reasons, we developed a minimal 32-bit microprocessor architecture μMD for fast collision search. It uses a very small instruction set, consisting of sixteen native commands, see Table 5 in the Appendix. In particular, this is sufficient for the execution of all algorithms of the MD4-family. Moreover, it suffices for the execution of current and (probably) future collision search algorithms, like CS. For the choice of our instruction set we compacted the results from Section 4 and used a processor reference design for μMD based on [16]. Furthermore, we designed the instruction set to maximize reuse of program code wherever possible. As a result, we also implemented a sufficiently large hardware stack and indirect load and store operations. In combination, they provide a comfortable access to parameterized subfunctions.

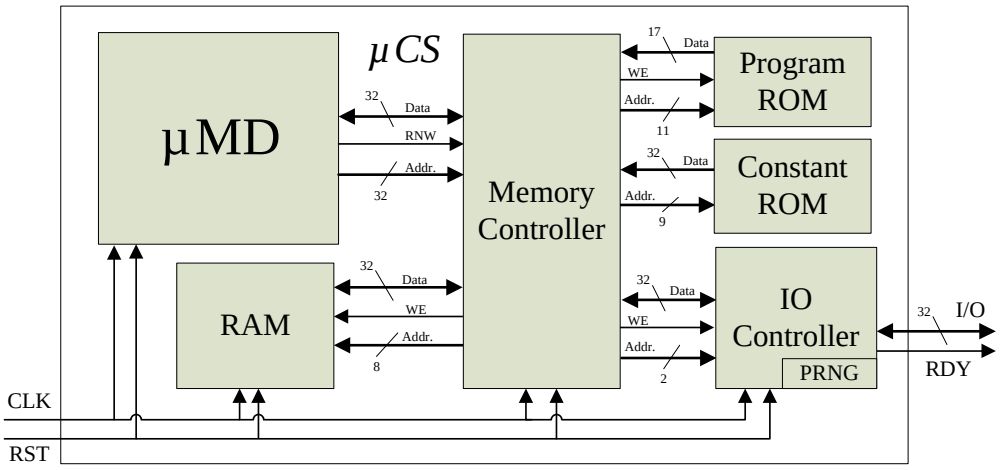

Figure 2: Simplified schematic of collision search unit μCS

5.3 Collision Search Unit

μCS is our final integrated circuit for collision search. Roughly spoken, it consists of a single μMD unit, additional memory and I/O logic. To start a computation, it requires an initial seed for the integrated PRNG. By carefully choosing this seed, we can guarantee that each μCS unit computes a different partition of the entire search space. When a collision is found, the corresponding message words are returned. Except for the PRNG initialization phase and the collision output sequence, there is no further I/O communication required. This decisively supports parallelization approaches making the overhead for additional control logic negligible. Figure 2 shows a simplified schematic of the collision search unit.

6 Implementation

Compared to general-purpose processors, μMD is small. μMD requires about 6k gate equivalents (GE), μCS roughly 210k GE. In an FPGA implementation, it uses about 9 percent of the slices of a low-cost Spartan3 XC3S1000 FPGA. For this device, the final clock frequency was reported with 95 MHz after synthesis.

6.1 Collision Search ASIC

Synthesizing μCS for standard cells (UMC 130 nm) requires 0.959618 mm^2 chip area. The sole processor μMD can be realized with only 0.026626 mm^2. This is just 2.77% of the chip area for the full μCS unit.
As we expected, the vast majority of chip area is used for the implementation of memory logic. For comparison reasons, we also tested collision search on a standard PC processor. We used a Pentium 4, 2.0 GHz machine (Northwood core) with approximately 55 million transistors build in 130 nm circuit technology [5]. Its die size is 146 mm^2.
We define time T as the average time for a single unit to find a collision. Unfortunately, μMD and μCS have not been built yet so simulated values will be used instead. For μCS this measure is computed based on the average number C of cycles required to find a collision and the

Architecture	**Cycles** C	**Frequency** f	**Period** t	**Time** T
μCS	$480{\cdot}10^9$ cycles	102.9 MHz	9.71 ns	4660.8 s
μMD	$480{\cdot}10^9$ cycles	228.8 MHz	4.37 ns	2097.6 s
Pentium 4	$60{\cdot}10^9$ cycles	2 GHz	0.5 ns	30 s

Table 1: Processor performance - average time T to find a collision

Architecture	**Time** T	**Area** A	$P = A \cdot T$
μMD	2097.6 s	0.026626 mm^2	55.9
Pentium 4	30 s	146 mm^2	4380

Table 2: Processor comparison for area-time product P

corresponding frequency f. Instead of f we can also use the reciprocal clock cycle period t.

$$T = \frac{C}{f} = C \cdot t$$

When implemented in our dedicated assembly language, the execution of CS in the complete simulation model is too inefficient to achieve reliable values. In the following, we will estimate the average number of clock cycles needed to find a collision for MD5. As a reference we use the average number of cycles needed by the Pentium 4 standard PC.
Although directly implemented in assembly, we believe the required average number of clock cycles for the execution of CS to be higher than that of CS. This is due to two major points.
Firstly, in contrast to standard PC processors, most μMD instructions consume two clock cycles. We assume that this fact roughly doubles the number of required clock cycles compared to the Pentium 4.
Secondly, unlike Pentium 4 systems, μMD does not make use of instruction pipelining for memory access operations. This means, that store and load operations cause the ALU of μMD to halt until their completion. On μCS, these operations are not only used to load and store data but also to fetch new instructions and to control the I/O including the PRNG. We believe this fact increases the number of required clock cycles by a factor of four.

6.2 Performance Comparison

Altogether, we estimate CS executed on μMD to require roughly eight times more clock cycles than on a Pentium 4. Assuming equal production constraints (same price per chip area, see Equation 1), each of our solutions is much more effective than a comparable Pentium 4 architecture when comparing the area-time product $P = A \cdot T$.
Table 2 compares the performance characteristics of μMD with those of a standard Pentium 4 processor. It is obvious, that μMD has better characteristics than the Pentium 4 processor. The performance of μMD for collision search is about 62 times better than the Pentium's. When we use an off-the-shelf standard PC for parallelization, we have to consider the costs for a motherboard with a network card that supports Preboot Execution Environment (PXE) ($\approx$ 80 €), fan ($\approx$ 12 €), RAM ($\approx$ 25 €), power supply ($\approx$ 25 €), additional equipment for the network infrastructure (network cables, switches), and a control server. Using these standard PCs, the processors are connected to each other by standard network equipment. Altogether, we believe the costs to be approximately 200 €, whereas we assume the Pentium 4 2.0 GHz to have a price of roughly 50 € leading to an overall parallelization overhead O_p of 300 %. From a Pentium 4

Metric	Pentium 4	μCS
Costs per area Q_A	$0.3425\frac{€}{mm^2}$	$0.3425\frac{€}{mm^2}$
Chip area A	146 mm^2	0.959618 mm^2
Chip cost Q_s	50 €	0.3287 €
Overhead O_p	300 %	5 %
System cost Q_p	200 €	0.3451 €
AT-Product P	4380	4472.6
Time T per unit	30 s	4660.8 s
Costs R for a collision per second	6000 €	1608.4 €

Table 3: Comparison of Pentium 4 and μCS architectures

Architecture	Costs R for a collision per second	Ratio R/R_{P4}
Pentium 4 PC	6000 €	100 %
μCS	1608.4 €	26.8 %

Table 4: Performance ratio of μCS compared to Pentium 4

processor, we estimate the price per chip Q_A area to be

$$Q_A = \frac{50\ €}{146\ mm^2} = 0.3425\frac{€}{mm^2}\ . \tag{1}$$

The parallelization of μCS however, requires only few additional logic per unit. In combination with low-throughput bus connection of μCS units, this provides an optimal scaling solution without noticeable additional costs. For our solution, we assume the area overhead to be almost negligible (less than 5%).
Based on these considerations, we believe that our *full* collision search solution is noticeably more effective for collision search than parallelized Pentium 4 processors.
Our estimates are summed up in Table 3. Q_s reflects the price for a single standalone unit of the corresponding architecture. In contrast, Q_p is the average price for a single unit after parallelization.
Obviously, for finding a single MD5 collision per second one has to spend $R = 6000$ € in (30) parallelized standard PCs (see Figure 3). Assuming similar constraints for manufacturing ASICs and excluding any NRE costs, the asset cost for the same performance invested in parallelized μCS units is $R = 1608.4$ €. So, it is almost four times more cost-efficient than the Pentium 4 standard PC architecture (see Table 4).

6.3 Estimations for SHA-1

We believe that a comparable implementation of a SHA-1 collision search algorithm in dedicated and parallelized collision search units has even better performance characteristics. This is mainly due to the fact, that SHA-1 needs much less constants than MD5, thus radically reducing the costs for (constant) ROMs. We assume that a collision search algorithm for SHA-1 can be programmed similarly compact. Although SHA-1 spans more steps, what is surely also reflected in the corresponding collision search algorithm, the program code will not considerably reflect this. This can be concluded from the unique implementation of subroutines which can be called on demand using only few additional instructions.
The average number of required clock cycles to find a collision is primarily dependent on the available theoretical results. Currently, attacks on SHA-1 have a complexity of about 2^{62} compression function evaluations. For practical attacks, we believe this number to be still very large.

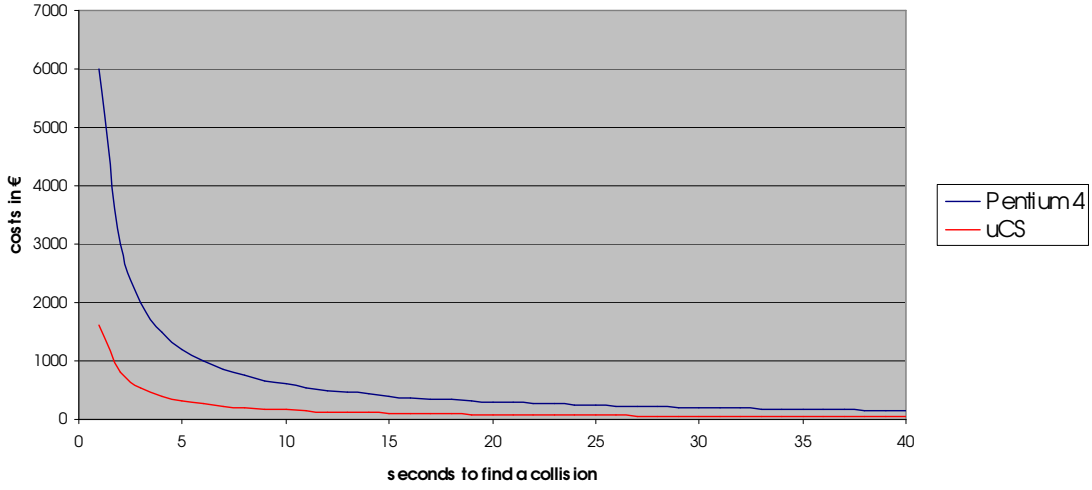

Figure 3: Costs for equipment to find a MD5 collision per time

Given 1,000,000 €, an attacker can buy enough standard PC equipment to find a MD5 collision within $\frac{6000}{1000000} = 0.006$ s on average. Invested in our collision search unit it would take only $\frac{1608.4}{1000000} = 0.0016084$ s. Finding MD5 collisions based on CS has been reported in [7] to have a complexity of about $7.7 \cdot 2^{30}$ MD5 step operations. The current bound for SHA-1 collisions is 2^{62} compression function evaluations, while each such evaluation is composed of 80 step function evaluations. Assuming an actual complexity bound of 2^{70} step operations and a similar execution time for a single step operation in MD5 and SHA-1, finding collisions for SHA-1 takes about 2^{37} times more time than for MD5. We can also conclude how long it would take to find a single SHA-1 collision with equipment for 1,000,000 €. Invested in standard PCs, it would take $0.006 \text{ s} \cdot 2^{37} \text{ s} \approx 26$ years. Using our collision search units, this time would be only $0.0016084 \text{ s} \cdot 2^{37} \text{ s} \approx 7$ years. Assuming Moore's law to hold for the next years, a successful attack for 1 million € in one year using a parallel μCS architecture should be possible in 2012 (less than next five years).

7 Conclusion

In this work we analyzed the hardware requirements of current and future collision search algorithms for hash functions of the MD4-family. We used our results to develop an appropriate hardware platform.

The heart of our design is a very small microprocessor μMD with only sixteen instructions. At the same time, it provides very effective means to support program code reuse, what greatly helps to keep the size of our overall collision search unit μCS small.

In the context of MD4-family hash functions, μMD is general-purpose, meaning that it is appropriate for the execution of all (32-bit) MD4-family hash functions and also of all corresponding current and future collision search algorithms.

In contrast to standard PCs, the final collision search unit needs only very little additional logic. This reduces its price and greatly eases parallelization approaches.

We believe that our design approach is much better suited for collision search than standard PCs. When money is spent on collision search, our design, massively parallelized, is nearly four times more cost-efficient than parallelized P4 standard PCs.

Appendix: Instruction Set for μMD

Opcode	Hex. value	z-flag	cycles	Description
RL	0000	not altered	2	Rotate A's bits to the left. The rotation width is found at the specified memory address.
STA	0001	not altered	2	Store A to absolute memory or IO address
STI	0010	not altered	3	Store A to indirect memory or IO address
LDI	0011	not altered	3	Load A from indirect memory or IO address
LDA	0100	not altered	2	Load A from absolute memory or IO address
ADD	0101	possibly altered	2	Add $\left(\mod 2^{32}\right)$ specified memory word to A
SUB	0110	possibly altered	2	Subtract $\left(\mod 2^{32}\right)$ the specified memory word from A
OR	0111	possibly altered	2	Compute logical OR of A and specified memory word
AND	1000	possibly altered	2	Compute logical AND of A and specified memory word
XOR	1001	possibly altered	2	Compute logical XOR of A and specified memory word
JMP	1010	not altered	2	Jump to specified address
JE	1011	not altered	2	Jump to specified address if z-flag is set to '0'
JNE	1100	not altered	2	Jump to specified address if z-flag is not set to '0'
CALL	1101	not altered	2	Push incremented program counter onto the stack and jump to specified address
RET	1110	not altered	3	Jump to address stored in top of stack. Pop top of stack
NOT	1111	possibly altered	1	Compute logical NOT of A

Table 5: Instruction set of μMD processor

References

[1] E. Biham and R. Chen. *Near-Collisions of SHA-0.* In *Advances in Cryptology — CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer Verlag, August 2004.

[2] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*, 1993.

[3] C. De Cannière, F. Mendel, and C. Rechberger. *On the Full Cost of Collision Search for SHA-1.* Presentation at ECRYPT Hash Workshop 2007, May 2007.

[4] C. De Cannière and C. Rechberger. *Finding SHA-1 Characteristics: General Results and Applications.* In *Advances in Cryptology - ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer Verlag, December 2006.

[5] Intel Corporation. *Intel Pentium 4 Processor Specification Update*, May 2007. Available for download at `www.intel.com`.

[6] M. Daum. *Cryptanalysis of Hash Functions of the MD4-Family.* PhD thesis, Ruhr-Universitt Bochum, 2005. Available for download at `http://www.cits.rub.de/MD5Collisions/`.

[7] D. Joŝĉák. Finding collisions in cryptographic hash functions. Master's thesis, Univerzita Karlova v Praze, 2006.

[8] V. Klima. *Project Homepage.* `http://cryptography.hyperlink.cz/MD5_collisions.html`, 2006.

[9] V. Klima. *Tunnels in Hash Functions: MD5 Collisions Within a Minute.* Cryptology ePrint Archive, Report 2006/105, 2006. Available for download at `http://eprint.iacr.org/`.

[10] A. Lenstra and B. de Weger. *On the possibility of constructing meaningful hash collisions for public keys.* ACISP, 2005.

[11] A. Lenstra, X. Wang, and B. de Weger. *Colliding X.509 Certificates*, 2005. Available for download at `http://eprint.iacr.org/`.

[12] J. Liang and X. Lai. *Improved Collision Attack on Hash Function MD5.* Cryptology ePrint Archive, Report 2005/425, November 2005. Available for download at `http://eprint.iacr.org/`.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, USA, 1997.

[14] R. Merkle. *One Way Hash Functions and DES.* In *Advances in Cryptology — CRYPTO '90*, volume 435 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[15] I. Damgård. *A Design Principle for Hash Functions.* In *Advances in Cryptology — CRYPTO '90*, volume 435 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[16] J. Reichardt and B. Schwarz. *VHDL-Synthese.* Oldenbourg, third edition, 2003.

[17] R. Rivest. *The MD5 Message-Digest Algorithm, Request for Comments (RFC) 1321*, 1992.

[18] Y. Sasaki, Y. Naito, N. Kunihiro, and K. Ohta. *Improved Collision Attack on MD5.* Cryptology ePrint Archive, Report 2005/400, November 2005. Available for download at `http://eprint.iacr.org/`.

[19] M. Stevens. *Fast Collision Attack on MD5.* Cryptology ePrint Archive, Report 2006/104, 2006. Available for download at `http://eprint.iacr.org/`.

[20] X. Wang, Y. L. Yin, and X. Yu. *Finding Collisions in the Full SHA-1.* In *Advances in Cryptology — EUROCRYPT 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer Verlag, August 2005.

[21] X. Wang and X. Yu. *How to Break MD5 and other Hash Functions.* In *Advances in Cryptology — EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer Verlag, May 2005.

[22] Xiaoyun Wang. Cryptanalysis on hash functions. Presentation at Information-Technology Promotion Agency (IPA), Japan, October 2006. Available for download at `http://helppc.jp/security/event/2006/crypt-forum/pdf/Lecture_4.pdf`.

[23] G. Yuval. *How to Swindle Rabin. Cryptologia*, Vol. 3(3):187–189, 1979.