Complexity Estimates of a SHA-1 Near-Collision Attack for GPU and FPGA

Stefan Gradinger
Bernhard Greslehner-Nimmervoll
Research Group Sichere Informationssysteme
FH OÖ Research and Development
Hagenberg, Austria
Email: stefan.gradinger@fh-hagenberg.at
Email: bernhard.greslehner-nimmervoll@fh-hagenberg.at

Jürgen Fuß
Robert Kolmhofer
Dept. f. Sichere Informationssysteme
University of Applied Sciences Upper Austria
Hagenberg, Austria
Email: juergen.fuss@fh-hagenberg.at
Email: robert.kolmhofer@fh-hagenberg.at

***Abstract*—The complexity estimate of a hash collision algorithm is given by the unit *hash compressions*. This paper shows that this figure can lead to false runtime estimates when accelerating the algorithm by the use of graphics processing units (GPU) and field-programmable gate arrays (FPGA). For demonstration, parts of the CPU reference implementation of Marc Stevens' SHA-1 Near-Collision Attack are implemented on these two accelerators by taking advantage of their specific architectures. The implementation, runtime behavior and performance of these ported algorithms are discussed, and in conclusion, it is shown that the acceleration results in different complexity estimates for each type of coprocessor.**

***Keywords*-SHA-1, hash function, hash collisions, near-collision, GPU, FPGA**

I. Introduction

Even though SHA-1 has been regarded as insecure for some time now [2], it is still in widespread use. All of the major web browsers as well as many applications still accept certificates that are signed with SHA-1. Many of the global root certificates also use it as part of their signature algorithm and those are often valid until the late 2020s.

This paper will therefore take a closer look at the current state of SHA-1 collision searches. Although the complexity of attacks on SHA-1 keeps decreasing, no one has yet found a full-collision (over 80 rounds), with the strongest successful attack attaining a collision over 75 rounds [1] and the shortest theoretical attack having a complexity of 2^{61} SHA-1 hashes [8].

After a brief evaluation of the currently available algorithms for collision search, the focus of this work will lie on the algorithm with the lowest complexity. This algorithm will be explained briefly and suitable parts will be implemented for the GPU and FPGA coprocessors with the languages OpenCL and VHDL, respectively. Implementation details as well as the performance will be explained and the runtime in a small high-performance cluster evaluated. Concluding we will take a look at the complexity of the algorithm and how the usage of an accelerator influences it.

II. SHA-1 and Collision Resistance

SHA-1 is a one-way compression function generating a 160-bit, random-looking output—called hash. As can be seen in Figure 1, the inputs of this hash function are a 160-bit initial vector (IV) and the message (Msg) to be hashed. SHA-1 extends the message, so that its length is always multiple of 512 bit. Multiple SHA-1 blocks are chained if the message exceeds the 512-bit boundary. In this case the hash value from the previous SHA-1 block is taken as the IV for the next block. The first block, however, gets standardized 32-bit values as IV input. All SHA-1 operations such as rotate, addition, and various boolean operations work on 32-bit integers, so input and internal state values are deemed 32-bit integers.

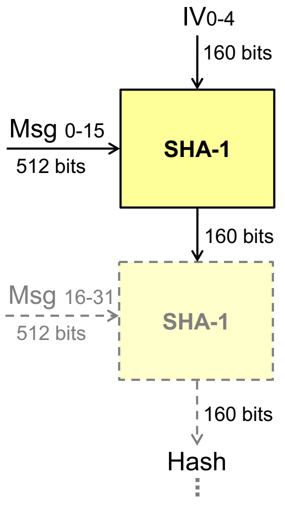

Figure 1. Two SHA-1 blocks in a chain and their input and output ports.

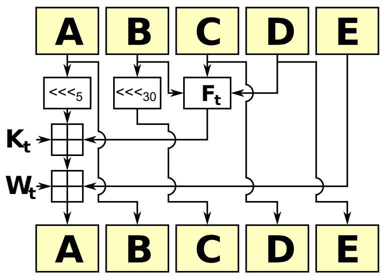

Figure 2. Block diagram of one SHA-1 round.

Figure 2 shows the internal structure of one SHA-1 round. A, B, C, D, and E are the internal 32-bit states. F_t performs boolean operations on B, C, and D. K_t is a predefined round constant and W_t is the expanded message for a specific round. The next A is calculated by performing four modulo-2^{32} additions of "A rotated left 5", F_t, K_t, W_t, and E. A full SHA-1 block is designed to repeat this procedure for 80 rounds.

One of the most important attributes of SHA-1 in specific and hash functions in general is their collision resistance. Collision resistance means that it is unfeasible to find two different input messages which produce the same output. This attribute is crucial for signature algorithms, as it prevents the forging of signatures. For performance reasons documents and data are usually not signed directly, instead the much smaller hash of the document is signed. If it were possible to forge a document which produces the same hash as an already signed document (collision), the attacker could use the signature of the signed document to legitimate his own forgery. Due to this critical importance, collision resistance is also the attribute which gets attacked most.

III. Related Work and Collision Algorithm

The first attacks on SHA-1 collision resistance were published by Wang et al. [10] in 2005 and had a complexity of 2^{69} full SHA-1 compressions. (In this paper the unit of complexity is "full SHA-1 compressions" unless explicitly stated otherwise.) In the following years a flurry of research activity followed which all had a lowered complexity as their main goal. Although some claimed a radically lowered complexity [9], down to 2^{60} [7] and even 2^{58} [4] most of these results could not be verified. Further advances were made at reduced SHA-1 functions with collisions found on CPU up to round 70 [3]. With the help of cryptographic coprocessors, these results have been expanded to round 75 [1] on graphics processing units (GPU). For field-programmable gate arrays (FPGA) algorithms have been designed for 71 and 75 rounds which claim lowered costs by an order of magnitude [5]. The public algorithm with the currently lowest complexity is from Marc Stevens [8] with 2^{61} for an identical-prefix collision and 2^{64} for a chosen-prefix collision. An identical-prefix collision has the additional limitation that both collision messages must have the same initialization vector while the IV can be freely chosen at the chosen-prefix collision.

Hashcat reports SHA-1 benchmark results for the AMD Radeon R9 290X, which is capable of $2^{31.9}$ hashes/s. For the FPGA, a fully pipelined SHA-1 implementation was developed. Each SHA-1 core outputs one hash every clock cycle as soon as the pipeline is filled. For the FPGA on the Xilinx ML605 board, it is feasible to instantiate eight cores, which operate at a frequency of 125 MHz. This results in a SHA-1 throughput of $2^{29.9}$ hashes/s. On a Xeon E5 hexacore CPU benchmarking results in $2^{24.2}$ hashes/s. Looking at these numbers it seems reasonable to implement a near-collision search algorithm on GPU (speed), FPGA (speed and power consumption), or a hybrid cluster of different processors. The numbers in Section VI will show that this is not the case.

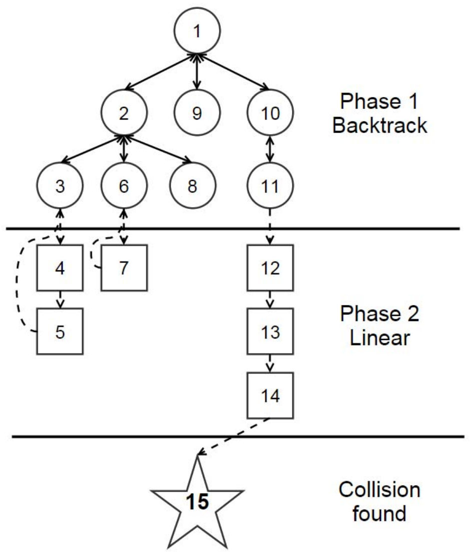

Figure 3. Diagram of the two phases and their relation with each other. The numbers in the circles and squares show the call order.

In addition to having the lowest complexity, Stevens' algorithm is also available as open source, which makes it the ideal candidate for portation to GPU and FPGA. Instead of going straight for the full collision, this work will focus on near-collisions which are a prerequisite to the identical-prefix collision search and which have a lowered complexity of $2^{57.5}$. A near-collision attack is achieved when two message blocks produce a hash output which is still different, but only in specific, intended ways. Afterwards these hash outputs are used as input for the prefix collision search.

The collision search of Marc Stevens—from now on called Hashclash—consists mainly of two phases. The first phase is a backtracking depth-first search over the first 35 rounds of SHA-1. The parameters for this search and the conditions for the advancement in the algorithm are given by a differential path[1], which describes the required difference between two message pairs at the rounds 1 to 30. Should a message not fulfill the requirements at a certain round, the algorithm steps back to the previous round and tries a different variation. This continues until there are either no new variations possible, in which case a completely new message gets started, or the message exceeds round 35, where it gets passed to Phase 2 of the algorithm. The second phase of the algorithm is a linear hash calculation and message check. There are no variation possibilities and either a message fits the criteria or it does not. If a message fails at the second phase it is passed

[1]The path used by Hashclash differentiates from the path given in Stevens' paper [8]. This new path goes up to round 35 instead of 20.

back to the first phase where variations are once again possible. See Figure 3 for a graphic representation of the two phases.

In detail, these two phases contain multiple subphases. For the first phase these are called steps. While these steps correlate with the rounds of SHA-1, the correlation is not strict. A single step might span over multiple SHA-1 rounds and there might be multiple steps for a single round. In the original implementation there are 29 steps for the first 35 rounds of SHA-1 with the Step 16_33 spanning the most rounds, Step 18 and Step 18b covering only a single round, and Step 22_01 to Step 24_01 together with Step 22_4 to Step 24_4 covering the same rounds in multiple ways.

The second phase consists of three subphases which are situated right after each other and are implemented in the function CheckNC. Each of these subphases covers a multitude of rounds with the first being responsible for the rounds 33 to 52 and the second for the rounds 53 to 60. In order to pass these two subphases, certain variables of the internal state of both messages must match. The third and final subphase calculates the rounds 61 to 80 and checks the internal state. In addition to that, it also checks if the hash outputs match one of the 192 desired differential results. Should a message pair fail in any of those subphases, it is sent back to the non-linear first phase. There is no possibility of variation in the second phase which means that each of these subphases has a very small probability that a message pair will pass them. Summing up these probabilities the overall percentage of success in this phase can be calculated.

To calculate the required computation time for a successful near-collision search, the message generation speed of Phase 1 is multiplied by the success probability of Phase 2. This opens up two possible avenues for speedup. The first is a better differential path, which has a higher success chance at Phase 2. The second is the acceleration of Phase 1 so it generates candidates more quickly for Phase 2, which is what this work will focus on.

As shown in Figure 4, the portation is restricted to the steps 18 and up. The steps before are ill suited for computation on GPU and FPGA; additionally, only 15 % of the computational time is spent there, which makes a portation unappealing.

IV. Massively Parallel Implementation on a GPU

A. Architecture on the GPU

The original implementation of Hashclash uses a depth-first search to find near-collisions. With its low memory consumption and good caching behavior, it is a good fit for the CPU. On the GPU this search pattern has several drawbacks, the biggest being thread-divergence. Thread-divergence occurs when threads of a single wave take varying execution paths. (A wave is a unit of 32 threads which share a scheduler.) Most often this occurs when some threads in a wave evaluate an if-statement to true and others to false. When thread-divergence occurs, the scheduler picks one execution path and executes the threads on it. While it is doing that, all threads on the other execution path are paused. After the first execution path has finished, the second gets loaded and executed. With depth-first search thread-divergence would occur whenever a thread descends in the search and others stay behind.

The actual performance penalty would vary with the current state of execution. At the very beginning the performance penalty could still be small with only minor thread-divergence issues occurring. However, once the execution has passed Step 18b, a step rife with conditional statements, the consequences for the performance would be devastating, with only a small percentage of threads still working.

Due to this, the GPU implementation of Hashclash uses a hybrid implementation of depth-first and breadth-first search.

The breadth-first part of the search works as follows: Every step of the algorithm is computed separately by all threads of the GPU in concert, which is possible due to a shared input and output buffer. In order to compute a step, each thread gets assigned an input according to its ID, which is used to execute the step operations. Whenever a thread descends to the next step, it copies its current dataset to the output buffer (input buffer of the next step) instead, which may occur multiple times or never at all for a single input. Once all inputs of the current step have been computed, all threads of the GPU advance to the next step.

The depth-first part of the search is due to the fact that the search space for the collision (and thus the inputs for the algorithm) is nearly limitless. With unlimited memory the computation would be stuck on the first step without ever advancing through the algorithm. In order to prevent that, the input size for each algorithm iteration is limited, which ensures a fast depth-wise traversal of the search tree. Note that this limit only affects the input size for the first step. The buffers between the following steps are big enough to hold all step outputs, even when a single input produces multiple outputs.

While this approach eliminates thread-divergence between the steps, there are still sources within each step that cause split execution paths. Where it was feasible, steps have been rewritten in order to reduce or eliminate these sources. Sometimes thread-divergence has also been accepted as too difficult or too resource intensive to circumvent.

B. Implementation of the steps

The steps 18 to CheckNC are implemented on the GPU as individual kernels. This is necessary due to the fact that OpenCL 1.2 lacks global barriers, which are needed to confine the computation of the threads to a single step. To circumvent this limit, we use the fact that the computation on the GPU is synchronized after each kernel completion. By implementing the steps as individual kernels, all threads on the GPU are synchronized after each completed step. This allows them to execute the next step together.

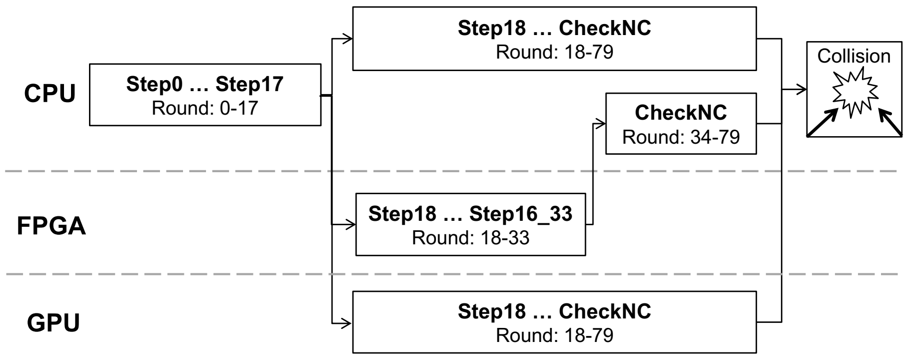

Figure 4. Architecture used in the near-collision search with the various coprocessors displayed.

All in all the steps are implemented in 22 kernels.

Communication and data exchange between the kernels is done via global memory. In total there are five input/output buffers, as well as 23 atomic counters. These counters hold the fill status of the buffers and are incremented every time a hash-state is written to them. A kernel which reads data from the buffer does not decrement or change the counter. Instead it uses its global thread ID to find the input which belongs to it.

The size of each buffer is 50 times greater than needed for the input of Step 18. These large buffers are required, as the size of the hash data will have increased 27 times once the computation reaches Step 19. With nearly twice the usually needed space the buffers should be able to hold even an above average increase. In the extraordinary event of running out of space, the additional output is discarded. While this is a drastic measure, it will not hurt the search overly much. Each output only has a marginal chance of being the one to find a near-collision and there are multiple outputs capable of doing so. Losing some outputs will slow down the computation, but it will not endanger the search itself.

When Step 18 is started with 14,060 input items with each item using 288 bytes, the total memory consumption for all buffers is about 965 megabytes.

C. Optimization

In order to speed up the computation multiple optimization strategies have been implemented:

- Hash-states are accessed using coalesced memory access. With this, memory operations of different threads can be done in parallel, which increases data throughput. Exempt from this are the input of Step 18 and the output of CheckNC. The GPU receives its input data for Step 18 in a format which does not allow coalesced access and writes it in the same format at CheckNC. Converting the data structure on the CPU into a fitting model has proven to be prohibitively expensive.
- Usage of the cl_ext_atomic_counters_32 extension. The standard global atomic operations are relatively slow. In order to speed up those operations, specialized atomic counters are used, which are capable of using the GPU's caching capabilities. Operations on these specialized counters execute about ten times faster than on the standard counters.
- Forgoing complete hash-state copies between the Steps 18b and 19. Between Step 18b and 19 only the changes in the internal hash-state are saved. This serves to reduce the amount of memory operations, since an input in Step 18b generates 27.5 inputs on average for the Step 19.
- Steps 18b, 19 and 21b have been split into multiple subkernels in order to reduce in-kernel thread-divergence. Every one of these steps has complex conditions which would incur a strong thread-divergence. Instead of executing divergent code lines, the IDs of the responsible inputs are stored. After the main execution path is finished, those IDs are recalled and processed in parallel by all threads. Since all executed inputs are now of the same path, no threads are paused.

D. Performance analysis

The performance analysis of the GPU code is done using the tool CodeXL from AMD, which captures all data points relevant to the execution.

Thread-divergence across all kernels is at 30 %, with the worst kernel being Step 18b part 3 at 60 % divergence.

The algorithm speed is bandwidth bound. Across most kernels the memory units are utilized 80–90 % with a total of 25 % stall time caused by fetch/write delays. The actual computational units are only used to 1.62 % of their capacity.

The total execution time for all 22 kernels with 14,080 Step 18 inputs on an AMD Radeon R9 290X is 10.04 milliseconds. The throughput of single GPU search instance is about 1,240,000 Step 18 inputs per second. The discrepancy between the time needed for a single execution run and the overall throughput per second is due to the overhead of scheduling and copying data to and from the GPU. On an Nvidia GeForce GTX 780 the throughput averages at 715,000 inputs per second.

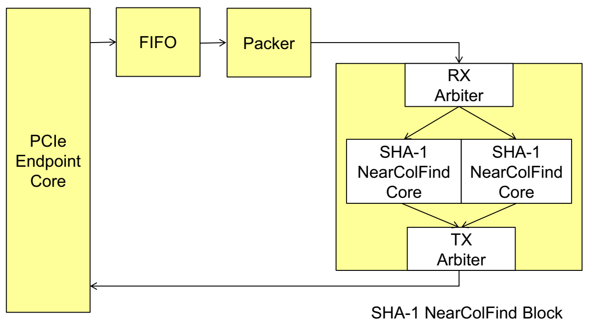

Figure 5. The SHA-1 NearColFind FPGA system.

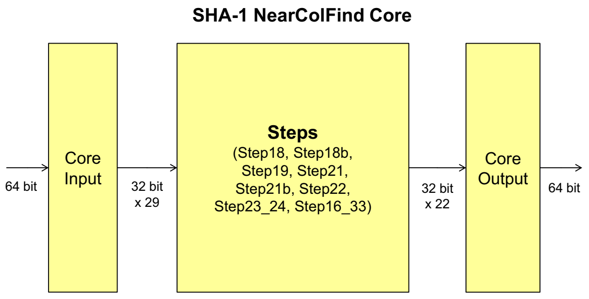
SHA-1 NearColFind Core
64 bit
Core Input
32 bit x 29
Steps (Step18, Step18b, Step19, Step21, Step21b, Step22, Step23_24, Step16_33)
32 bit x 22
Core Output
64 bit

Figure 6. Block diagram of SHA-1 NearColFind Core.

V. Massively Parallel Implementation on an FPGA

After analyzing the SHA-1 near-collision algorithm of Marc Stevens, all steps after Step 14 are identified as implementable on FPGA. Although it is possible to start from Step 15, Step 18 is chosen as point of entry on FPGA for the following reasons:

- Few function calls in comparison to other steps between Step 17 and Step 18.
- Most computation time is spent in Step 18b and Step 19.
- Uniform coprocessor outsourcing: Parallelization on GPU is not feasible until Step 18.

Figure 5 shows the whole FPGA system, which is implemented using the hardware description language VHDL. The FPGA system is designed to receive input data for Step 18 from the CPU and return the result of Step 16_33 to the CPU again. For CPU and FPGA communication, the PCI Express interface is used. On FPGA, the PCI Express IP core *PCI Endpoint Core* was developed with low FPGA resource consumption and high throughput in mind. After receiving the data on FPGA, it is buffered in a FIFO in order to equalize the latency between two PCI Express transactions. For system performance reasons, several Step 18 input data are sent in one PCI Express transfer which the *Packer* unit disassembles and forwards to the *SHA-1 NearColFind Block*. The *RX Arbiter* distributes the input data to several *SHA-1 NearColFind Cores*. Since the CheckNC calculation is done on CPU, the internal SHA-1 values of the collision calculation are sent back to CPU including a transaction ID (transID).

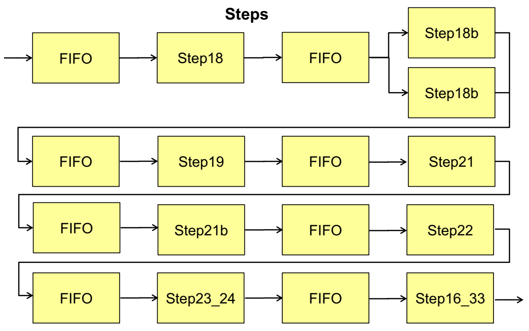

Figure 7. Data flow of SHA-1 NearColFind Steps.

Resource	Utilization	%
Registers	80,921	26
LUTs	69,718	46
Slices	26,297	69
RAM	288	69

Table I
FPGA resource utilization.

Figure 6 presents how the data width is converted before and after the unit *Steps*. The *Core Input* unit converts the 64-bit data stream from *PCI Endpoint Core* into 29 32-bit state values plus the transID. The *Core Output* unit serializes the output data including the transID to a 64-bit data stream which is sent back to CPU via the *PCI Endpoint Core*.

Figure 7 illustrates the organization of individual steps and the data flow through these steps and FIFOs. In the chain, each step presents a pipeline stage, so steps 18 to 16_33 can produce data for the next step at the same time. Individual steps generate data for the next in different numbers of clock cycles. The amount of generated input data for the next step depends on the current step and its input data. For that reason, a FIFO is placed before every step in order to equalize the different runtimes of each step. Thus it is possible to minimize the idle time for a step, provided that previous steps deliver enough input data. Furthermore, chosen steps can be instantiated several times in the chain (see Step 18b in Figure 7). This makes sense if the runtime of a specific step is longer than that of others.

The FPGA platform used is the Xilinx evaluation board ML605. For the Virtex-6 XC6VLX240T-1FFG1156 FPGA, two *SHA-1 NearColFind Cores* can be instantiated. The crypto cores operate at a frequency of 125 MHz. Table I shows the FPGA resource utilization. Under the CentOS 6.5 operating system, a throughput of 2.4 millions of Step 18 items per second could be achieved. It results in approximately 16,000 CheckNC function calls per second.

VI. Complexity and Runtime Estimates

The success probability p_l of the linear Phase 2 (rounds 32 to 79), obtained experimentally and determined to be $2^{-45.56}$ by Stevens [8], has been verified on our systems.

Coprocessor	Step 18/s	CheckNC/s
Intel Xeon E5 2620 CPU Core	140,000	960
FPGA Xilinx Virtex-6	2.4 mio	16,500
AMD Radeon R9 290X	1.2 mio	8300
Nvidia GeForce GTX 780	700,000	4800
Cluster Total	81 mio	563,000

Table II
HASHCLASH THROUGHPUT BY INDIVIDUAL COPROCESSORS AND TOTAL PERFORMANCE OF THE CLUSTER.

With this probability, the function CheckNC must be called $2^{45.56}$ times on average to get one SHA-1 near-collision candidate. To get the performance of the non-linear Phase 1 (rounds 0 to 35), the number of CheckNC function calls are counted and divided by the runtime in seconds.

A. Cluster Runtime Estimates

The cluster and its software framework [6] consists of 38 computing nodes containing 16 Xilinx ML605 FPGA boards, 27 AMD Radeon R9 290X and five Nvidia GeForce GTX 780 GPUs, and 239 Intel Xeon E5 2620 CPU cores.

Table II depicts the performance of each coprocessor type. The FPGA offers the best performance with 16,000 CheckNC/s. The AMD Radeon R9 290X is nearly twice as fast as the Nvidia GeForce GTX 780 with 8300 CheckNC/s, which is not surprising, as the portation is optimized for the AMD GPU. As expected, a single CPU core provides the least performance. However, the Intel CPU Xeon E5 has six physical cores, so the CPU can compute 5760 CheckNC/s if all its cores are used for SHA-1 computation. In practice the CPU also has to provide the input data for GPU and FPGA, which leaves fewer free cores for the last part of the computation.

The total CheckNC per second calls for the whole cluster are about $563,000 \approx 2^{19.10}$. Therefore, the estimated time required for finding one SHA-1 near-collision is $2^{45.56}/2^{19.10}\ s^{-1} = 2^{26.46}\ s \approx 2.93$ years.

The Hashcat project[2] reports SHA-1 benchmark results for the AMD Radeon R9 290X, which is capable of 3.94 billion hashes/s. For the FPGA, a fully pipelined SHA-1 implementation was developed. Each SHA-1 core outputs one hash every clock cycle as soon as the pipeline is filled. For the FPGA on the Xilinx ML605 board, it is feasible to instantiate eight cores, which operate at a frequency of 125 MHz. This results in a SHA-1 throughput of one billion hashes/s. Considering these numbers, such a long expected runtime for the near-collision search is unexpected. In theory, the required runtime can be calculated by dividing the Hashclash complexity of $2^{57.28}$ with the total hashing performance of the hardware. With the GPUs and FPGAs of the above mentioned cluster, which achieve a total performance of $2^{53.23}$ compressions per day, the near-collision search should only take about 16.5 days based on the complexity figure. In reality it would instead take the previously mentioned 2.93 years. The reason for this big discrepancy lies in the way the complexity is calculated.

[2]http://hashcat.net/oclhashcat

Coprocessor	Hashes/s	CheckNC/s	C_{nl}	Complexity
CPU Core	$2^{21.63}$	$2^{9.91}$	$2^{11.72}$	$2^{57.28}$
FPGA	$2^{29.90}$	$2^{14.01}$	$2^{15.89}$	$2^{61.45}$
GPU (AMD)	$2^{31.88}$	$2^{13.02}$	$2^{18.86}$	$2^{64.42}$

Table III
COMPLEXITY ESTIMAETS FOR VARIOUS COPROCESSOR TYPES.

B. Complexity Estimates

The complexity is measured in the unit SHA-1 compressions which is calculated by the formula C_{nl}/p_l, where C_{nl} is the average complexity of the non-linear phase and p_l is the success probability of the linear Phase 2. While p_l can be directly computed, Stevens calculates the complexity C_{nl} by comparing the time required to generate a CheckNC and the time required to do a full SHA-1 compression [8].

The problem here lies in the fact that the time required for a full SHA-1 compression is heavily dependent on the architecture of the coprocessor and on the implementation of the compression.

The complexity calculations in Table III indicate that the complexity figures differ strongly with the type of coprocessor used. For HashClash on the GPU you have to perform equivalently $2^{64.42}$ SHA-1 compressions whereas on CPU only $2^{57.28}$ are necessary to obtain a SHA-1 near-collision. Therefore, the figure for how many SHA-1 hashes can be computed per second has significant influence on the complexity calculation. Using a slow SHA-1 hashing implementation would actually reduce the complexity. In conclusion, it stands to reason that the unit equivalent SHA-1 compressions may not be the most suitable unit to depict the effort needed for a collision. This is doubly so, as the search for a collision resembles much more a backtracking tree search than a multitude of hash compressions.

VII. CONCLUSION

We have given a short introduction to SHA-1 as well as its main security attribute, collision resistance, and the main research pertaining to these topics. The GPU and FPGA implementations have been described, the optimization techniques illustrated, and the performance analyzed. Furthermore, we have shown runtime figures from a small HPC Cluster. Even though the estimated runtime is multiple years and thus out of reach we still hope that our result and methods can be used to further the research in the collision resistance field.

At the end we have taken a look at the problematic complexity calculation algorithm. We have shown that solely using the complexity for runtime estimations may be misleading as the complexity can vary greatly with the use of coprocessors and even with different implementations.

For future research it may be worth looking into different runtime estimation figures, which better reflect the practical computational effort required for finding a collision. Additionally, there is of course the research to reduce the computational effort needed for a successful collision search.

ACKNOWLEDGMENT

This study has been funded by the Austrian security research programme KIRAS of the Federal Ministry for Transport, Innovation and Technology (bmvit).

REFERENCES

[1] Andrew V. Adinetz, Evgeny A. Grechnikov, Building a Collision for 75-Round Reduced SHA-1 Using GPU Clusters, Euro-Par 2012 Parallel Processing, Lecture Notes in Computer Science Volume 7484, 2012, pp. 933–944

[2] Elaine Barker and Allen Roginsky, Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, NIST Special Publication 800-131A , pp. 13

[3] Christophe De Cannire, Florian Mendel, Christian Rechberger, Collisions for 70-Step SHA-1: On the Full Cost of Collision Search, Selected Areas in Cryptography, Lecture Notes in Computer Science Volume 4876, 2007, pp. 56–73

[4] Rafael Chen, New Techniques for Cryptanalysis of Cryptographic Hash Functions, Ph.D. thesis, Technion, Aug 2011

[5] Alessandro Cilardo, Nicola Mazzocca, Exploiting Vulnerabilities in Cryptographic Hash Functions Based on Reconfigurable Hardware, IEEE Transactions on Information Forensics and Security, 2013, pp. 810–820

[6] Danczul, B. and J. Fuß and S. Gradinger and B. Greslehner-Nimmervoll and W. Kastl and F. Wex, Cuteforce Analyzer: A Distributed Bruteforce Attack on PDF Encryption - In Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES), Regensburg, Germany, 2013, pp. 720–725

[7] Florian Mendel, Christian Rechberger, and Vincent Rijmen, Update on SHA-1, Rump session of CRYPTO 2007, 2007. Available online at http://rump2007.cr.yp.to/09-rechberger.pdf.

[8] Marc Stevens, New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis, In Proceedings of EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26–30, 2013, pp. 245–261

[9] Xiaoyun Wang, Andrew C. Yao, and Frances Yao, Cryptanalysis on SHA-1, NIST Cryptographic Hash Workshop Presentation, 2005

[10] 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.