Acceleration Attacks on PBKDF2
Or, what is inside the black-box of oclHashcat?

Andrew Ruddick *
Oxford, UK
andrew.ruddick@hotmail.co.uk

Jeff Yan
School of Computing & Communications
Lancaster University, UK
jeff.yan@lancaster.ac.uk

Abstract

The Password Based Key Derivation Function v2 (PBKDF2) is an important cryptographic primitive that has practical relevance to many widely deployed security systems. We investigate accelerated attacks on PBKDF2 with commodity GPUs, reporting the fastest attack on the primitive to date, outperforming the previous state-of-the-art oclHashcat. We apply our attack to Microsoft .NET framework, showing that a consumer-grade GPU can break an ASP.NET password in less than 3 hours, and we discuss the application of our attack to WiFi Protected Access (WPA2).

We consider both algorithmic optimisations of crypto primitives and OpenCL kernel code optimisations and empirically evaluate the contribution of individual optimisations on the overall acceleration. In contrast to the common view that GPU acceleration is primarily driven by massively parallel hardware architectures, we demonstrate that a proportionally larger contribution to acceleration is made through effective algorithmic optimisations. Our work also contributes to understanding what is going on inside the black box of oclHashcat.

1 Introduction

The Password Based Key Derivation Function v2 (PBKDF2) [20, 12] is an important cryptographic primitive, employed in a large number of widely deployed systems, such as WiFi Protected Access (WPA/WPA2), Microsoft .NET framework, Apple OSX Operating System user passwords, Apple iOS passcodes, Android passcodes (v3.0 – v4.3), Blackberry's password manager and Cisco IOS type 4 passwords, amongst many others.

Based on an underlying cryptographic hash function, e.g. SHA1 or MD5, PBKDF2 is used to derive a deterministic, cryptographically secure key from a given password and random salt. Since the derivation is closely related to the underlying hash function, PBKDF2 is applicable to any traditional application of cryptographic hash functions, such as secure password storage, as an encryption key for symmetric ciphers such as 3DES or AES, or for integrity protection, when used as a MAC.

In this paper, we accelerate PBKDF2 with the OpenCL framework and General Purpose Programming on Graphics Processing Units (GPGPU). Cracking or accelerating cryptographic primitives with GPGPU technologies is not new, but we address the following novel issues:

1. What are the limits of acceleration on cheap, commodity GPUs? In the process of pushing the limits, what new and deep insights can be learned?
2. The relative contributions of various optimisations on the overall acceleration process. There are various algorithmic optimisations for the primitives and various optimisation strategies for GPGPU programming, but it is currently unclear which of them contribute most to the overall acceleration process.
3. OclHashcat [17] reports for many primitives the best GPGPU acceleration, including PBKDF2. However, due to its closed-source nature, it is unclear what is going on inside this black box. We tackle the following interesting research questions:
   (a) Why does oclHashcat outperform competitors?
   (b) Are there hidden cryptographic vulnerabilities exploited by oclHashcat?
   (c) Can we improve its acceleration?

We first cover necessary technical background, then highlight both relevant algorithmic and kernel optimisations, assessing the individual contributions of various optimisation techniques on the acceleration. We report

*This work was done while as a full-time student supervised by Jeff Yan.

the fastest GPGPU accelerated attack on PBKDF2 to date. We present a practical attack on the Microsoft .NET framework implementation of PBKDF2, showing that a consumer GPU can crack a password in less than 3 hours, a worrying result for the security of this popular and critical framework. We then discuss the practical application of our work to the attack of WPA2. Additionally, our contributions also include providing a fine summary of how a cryptographic algorithm should be optimized for attack, and bridging the gap between academia and the password cracking community.

Academic research in this area was impeded by the topic's complex nature, the amount of engineering effort required and the lack of open-source efforts. As such, we release our source code (10,000+ lines of C and C++) at `https://github.com/OpenCL-Andrew/.NETCracker/` for further advancement of research in this field.

2 Technical Background

2.1 Cryptographic Primitives

PBKDF2 is a key stretching algorithm, which allows us to expand a key into n-bit output blocks. The algorithm allows for an arbitrary number of iterations to be set, so that over time, the algorithm's computational complexity can be increased. The standard is defined by NIST [20] and IETF [12], as follows:

$$P = \text{Password bytes; } S = \text{Salt bytes}$$
$$i = \text{Block ID; } c = \text{Iterations}$$
$$hLen = |HMAC(S,P)| = \text{ (160 bits for SHA1)}$$
$$dkLen = \text{Derived Key Length (bits)}$$
$$l = \lceil \tfrac{dkLen}{hLen} \rceil; r = \Big(dkLen - \big(l-1\big) \cdot hLen\Big)$$
$$int(x) = \text{32-bit Big-Endian representation of } x$$
$$X_{<n...m>} = \text{0-indexed substring range from } n \text{ to } m \text{ over } X$$
$$||\text{ denotes string concatenation}$$

$$U_{rc} = \begin{cases} U_1 = HMAC(P,S\,||\,int(i)) & \text{1st iteration} \\ U_2 = HMAC(P,U_1) & \text{2nd iteration} \\ \vdots & \vdots \\ U_c = HMAC(P,U_{c-1}) & \text{Final Iteration} \end{cases}$$

The round computations U_{rc} are used in the core repeated hashing, as defined by the function: $F(P,S,c,i) = \big(U_1 \oplus U_2 \oplus \ldots U_c\big)$. F is invoked for each required block of the derived key T_i, where the number of iterations l is defined by the required length of the derived key and is concatenated with the output of any previous block round computations: $T_i = F\big(P,S,c,i\big)$. The final block may be partial, dependent upon the specified size of the derived key length $dkLen$, so the PBKDF2 function is defined as:

$$PBKDF2(P,S,c,dkLen) = (T_1\,||\,T_2\,||\,\ldots\,||\,T_{l<0...r-1>})$$

HMAC (Hashed Message Authentication Code) is commonly defined as follows [6, 13].

$$K = \text{Secret Key; } M = \text{Message; } B = \text{Block Size (bytes)}$$
$$ipad = \text{Inner Padding (fixed byte 0x36, repeated } B \text{ times)}$$
$$opad = \text{Outer Padding (fixed byte 0x5C, repeated } B \text{ times)}$$
$$H = \text{SHA1; } B = 64 \text{ (SHA1 block size)}$$
$$HMAC(K,M) = H\Big((K \oplus opad)\,||\,H\big((K \oplus ipad)\,||\,M\big)\Big)$$

SHA1 produces a 20 byte output hash of an arbitrarily sized input value $\{0,1\}^{160} \leftarrow \{0,1\}^*$, as defined by NIST [7] and the IETF [10]. Its core compression function operates over input blocks of 512 bits, using Merkle-Damgård block construction to chain the compression of larger inputs. Figure 1 demonstrates the chained structure of SHA1, where h represents our compression function. The final block is length padded to ensure it consumes the full 512 bits. The binary number 1 is directly appended to the message bit sequence, followed by 0's until the last 64 bits, which are reserved for a representation of the message size. The maximum input to SHA1 is therefore 2^{64}.

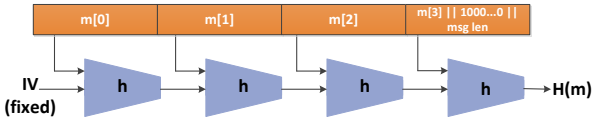


Figure 1: Merkle-Damgård Block Construction

Each block of the padded 512-bit message is processed in turn, first packed into 16 32-bit integer values, denoted $W[0],\ldots,W[15]$. An expansion function expands them to 80 32-bit words. $W[16],\ldots,W[79]$ are computed as follows:

$$\vee = \text{ bitwise inclusive OR; } \oplus = \text{ bitwise exclusive OR (XOR)}$$
$$<<= \text{Left logical bitshift; } >>= \text{Right logical bitshift}$$
$$ROTL(a,n) = ((a << n) \vee (a >> (32-n)))$$
$$W[t] = ROTL(W[t-3] \oplus W[t-8] \oplus W[t-14] \oplus W[t-16],1)$$

The function h is then built from a series of 80 Davies-Meyer compressions, defined as follows:

$$f(t,B,C,D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & (0 \le t \le 19) \\ B \oplus C \oplus D & (20 \le t \le 39) \\ \big((B \wedge C) \vee (B \wedge D) \vee (C \wedge D)\big) & (40 \le t \le 59) \\ B \oplus C \oplus D & (60 \le t \le 79) \end{cases}$$

5 32-bit integer variables are set to constant values (represented in hexadecimal):

$$H0 = 0x67452301; H1 = 0xEFCDAB89$$
$$H2 = 0x98BADCFE; H3 = 0x10325476$$
$$H4 = 0xC3D2E1F0$$

Then the 80 main round-stage computations are computed:

$$K(t) = \begin{cases} 0x5A827999 & (0 \leq t \leq 19) \\ 0x6ED9EBA1 & (20 \leq t \leq 39) \\ 0x8F1BBCDC & (40 \leq t \leq 59) \\ 0xCA62C1D6 & (60 \leq t \leq 79) \end{cases}$$

$$Round(t) = \begin{cases} TEMP = ROTL(5,A) + f(t,B,C,D) \\ \qquad +E + W[t] + K(t) \\ E = D \\ D = C \\ C = ROTL(30,B) \\ B = A \\ A = TEMP \end{cases}$$

The function $Round(t)$ is executed for $t = 0, \ldots, 79$, after which, the variables H0, ..., H4 are updated: $H0 = H0 + A, \ldots, H4 = H4 + E$. If there are multiple message blocks to process, the W storage array is re-initialised to the next 512 bits and the word-expansion phase re-executed, the 80 round-stage computations are calculated and the variables H0, ..., H4 updated. The final result is the 160-bit concatenation of $H0||H1||H2||H3||H4$.

Figure 2 gives a pictorial representation of the round-stage computations, where $f(t,B,C,D)$ is represented by F.

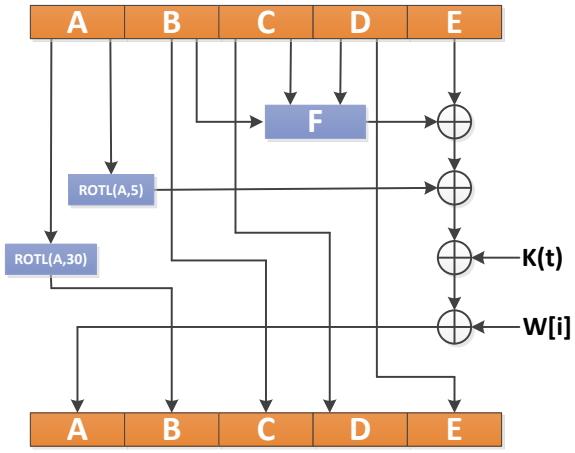


Figure 2: Davies-Meyer Construction

2.2 GPGPU Programming

We opted to use OpenCL over CUDA as it is more widely applicable. OpenCL code may be executed on many device architectures, including NVIDIA and ATI GPUs, Altera FPGAs [8] and Intel and AMD CPUs.

All of the GPGPU programming concepts in this section apply to both OpenCL and CUDA, though the terminology is focused on OpenCL – there are some subtle differences under the CUDA programming model.

The GPU and CPU fundamentally differ in architecture. A GPU has a slower clock speed, focusing instead on the ability to process many simultaneous streams of data through its Stream Processors (SIMD Vector Units). Typically, a GPU contains thousands of these SIMD-VU, making them ideal for computer graphics pixel-related calculations. GPUs are very well suited for problems that require FLOP-intensive computation and, provided that memory accesses are correctly aligned and coalesced, can offer massive memory bandwidth. However, a low processor register-count means that it can be hard to take advantage of the available memory throughput for all but the simplest problems. Here, we cover some key GPGPU programming concepts.

2.2.1 Data Buffering

In order to interface with the device you need to manually allocate memory buffer(s) on the GPU global memory storage, specifying both the size and type of memory, e.g. write only, read only or read and write (the slowest). Once allocated, the programmer must map an area of CPU global memory to the buffer and initiate a transfer over the system bus manually.

2.2.2 Kernel Programming

Once data has been loaded into the GPU global memory, custom programs can be executed to process the data. These programs are written in the form of GPU kernels, that first need to have source code compiled to assembly instructions by the device driver. This compilation needs to be done for each and every GPU device to run the code. Kernels can either be pre-compiled, or processed in a JIT fashion directly from the source code.

2.2.3 GPU Workload Division

SIMD-VU are designed to be able to perform a vector graphics operation over a pixel (x, y, z, w). Each unit has a number of dedicated hardware circuits able to directly perform various graphics processing operations. For cryptographic purposes, we are most interested in the fact that it is able to directly perform bit-shifting and bit-wise operations at the hardware level. More complex op-

erations, such as branches may result in code divergence within the wavefront, which will slow execution.

SIMD-VU are organised into a series of Compute Units (CUs) and the execution of a single kernel instance is termed a work-item, which is processed by a single SIMD-VU. Each CU physically consists of multiple processing elements, each of which contains a number of Arithmetic Logic Units (ALUs). Each ALU group may repeat each instructions execution over a number of cycles, to process a single logical execution unit, termed a wave, or wavefront. Kernel code executes in lock-step across all work-items in a given wave. Through multiple logical groupings of these processing elements, each CU can handle a number of wavefronts simultaneously, each of these groupings is termed a Work-Group (WG). It is worth mentioning however, that where Instruction Level Parallelism (ILP) is employed, it is equally possible that a given CU ALU group may have instructions from the same wavefront being processed in parallel, potentially using a pipeline. This makes ILP an important consideration for kernel code optimisation.

2.2.4 GPU Memory Management

There are three tiers of GPU memory, in increasing order of speed: global memory, Local Data Storage (LDS) memory and register memory. Programmers must manage this memory manually.

On AMDs Graphics Core Next (GCN) architecture, there is a 32-bank LDS per CU. As LDS has faster access speeds, a WG can copy data to LDS storage for processing by individual work-items. This helps reduce the length of time needed to access data as a result of memory stalls. If a shared data-reliance exists cross WG, within the same CU, global memory synchronisation is necessary.

Each CU has an allocation of general-purpose register memory that is shared amongst all processing elements within that CU. This is the fastest level of device memory and although on AMD GCN the register files are actually 4 times larger than the CU's LDS, heavy reliance on register memory will limit the number of processing elements that may operate on the data simultaneously; thus it is desirable to minimise register usage per work item whenever possible.

2.2.5 Memory Access Latency Hiding

Each CU handles multiple in-flight wavefronts at any time so that the large memory latencies experienced on GPU global memory accesses can be masked. A global memory operation generates a reference to the off-chip memory, causing a latency of between 300 and 600 cycles. The CU thread scheduler multiplexes the execution of multiple waves within each CU, so that this global memory latency can be hidden. When one active wave stalls as a result of an off-chip memory reference, another wave will be allowed to continue executing on that CU. As a result, it is important that the CU has a sufficient number of concurrently executing waves, so that all lengthy memory operations can be successfully masked.

As multiple concurrent wavefronts, spread across multiple Work-Groups execute within a single CU, the total available shared register count usage will bound the total number of wavefronts that can execute. Where high register usage is a constraint on the number of in-flight wavefronts that may be executed, utilisation of LDS bank storage and a slightly slower memory access may be a desirable trade-off.

3 Attack Design

Our attack has been designed to reduce typical GPGPU programming bottlenecks, specifically considering: minimising transfer of application data over the system bus, reducing register memory usage per work-item and efficient LDS storage use. Figure 3 shows a diagrammatic representation of the CPU-GPU interaction, where (m) is a password buffer, generated by the host.

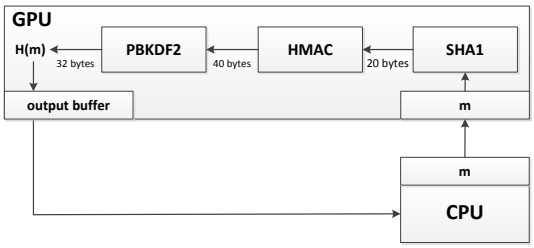


Figure 3: CPU-GPU Interaction

We reduce the latency of the data transfer by copying only a single buffer per GPU processing batch. It is assumed that all passwords loaded in a given block are of the same length, so dictionary attacks would have to be organised by password length using this scheme. Each Kernel operating on the GPU indexes into the password block, operating over a portion of its data. We have opted for this method instead of a GPU-side algorithm as it allows for both brute-force attacks over an arbitrary alphabet as well as dictionary based attacks.

The host application allocates a GPU constant buffer alongside the input buffer, which is the 32-bit uint representation of the target hash. After the hash for each work-item has been calculated, we check to see if it matches the target hash. If there is a collision, a single output register is set and the matching plain text password is stored. This requires we check only a single 32-bit output buffer

for each batch of hashes processed (unless there is a collision, when we must read 2 buffers). The trade-off is that the decrease in bus transfer time costs us additional cycles per work item instance to compute the collision on the GPU.

4 Optimisations

We consider both algorithmic optimisations of the cryptographic primitives and optimisations for GPU. We implemented all of the cryptographic algorithms directly following NIST and IETF specifications, rather than reusing existing reference implementations.

4.1 Algorithm Optimisations

4.1.1 SHA1

The first optimisation that we perform on SHA1 is redefining the 2,560-bit storage queue W to a cyclic queue of 512 bits, as per NIST FIPS 180-4 section 6.1.3 [7]. This method increases the computational complexity of addressing individual array elements, in favour of a smaller memory footprint, decreasing CU register pressure. The word expansion phase of SHA1 is removed and instead, accesses over $W[t]$ where $t \leq 15$ are re-defined as $W[t \wedge 0x0F]$ and where $(16 \leq t \leq 79)$ the following is used:

$$MASK = 0x0F$$

$$W[t \wedge MASK] = \begin{cases} ROTL((\\ W[((t \wedge MASK)+13) \wedge MASK] \oplus \\ W[((t \wedge MASK)+8\ ) \wedge MASK] \oplus \\ W[((t \wedge MASK)+2\ ) \wedge MASK] \oplus \\ W[(t \wedge MASK)]) \\ ,1) \end{cases}$$

The second optimisation is to the Merkle-Damgård construction. Allowing for the padding and representation of message size, we are left with 447 bits in the first message block for a candidate password, which is sufficient for up to a 55 character (ASCII) password. By removing the block nature of the algorithm, we save processing time. Thus, our SHA1 function becomes a series of 4 steps:

1. Length pad the password candidate
2. Initialise $H0, H1, H2, H3, H4$ to constant values
3. Compute $Round(t)$ for $t = 0, \ldots, 79$
4. Return $H0||H1||H2||H3||H4$

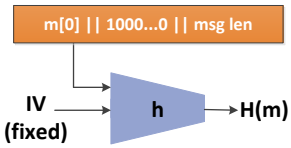


Figure 4: Modified SHA1 Merkle-Damgård Block Construction

Figure 4 demonstrates the optimisation over the reference implementation in figure 1.

The final optimisations are described in [19] and focus on reducing the instruction count per hash, exploiting information we know as a result of the above Merkle-Damgård optimisation.

- Initial step optimisations. As we are only processing a single block per hash, the first 3 $Round(t)$ operations contain a number of known values that can be pre-computed. In $Round(0)$, the only unknown is $W[0]$, therefore we can simplify this step to:

$$Round(0) = \begin{cases} A = (0x9FB498B3 + W[0]); \\ B = H0; \\ C = 0x7BF36AE2; \\ D = H2; \\ E = H3; \end{cases}$$

  In the same way, it is possible to optimise the following 3 initial steps, for a total reduction of 3 rotate operations and 7 additions.

- Zero based optimisations, which take advantage of the known structure of the message padding. For a password of up to 11 digits, we require only $W[0] - W[2]$ for storage, including the appended binary 1. The result is that W[t] is known to be $0x00000000$ for rounds 3 - 14, allowing for removal of an addition operation in the $Round(t)$ stages for these 11 iterations. The benefit depends upon the password candidate length.

- Early exit optimisations. Early exit allows us to recognise when we are calculating an incorrect hash candidate and abort further unnecessary calculations. We know the initial value of $H0 - H4$ for each candidate. The final SHA1 step sets the last 32-bit output block to $H4 = H4 + E$, so we first subtract $H4$ from the last block of our target hash (denoted $H4'$), then because the final two rounds, 78 and 79 of $Round(t)$ do not alter the final block we can exit after round 77, if $E \neq H4'$. However, also reversing the rotate operation performed in round 77, allows this early exit after step 75. This requires definition

of the rotate right function and a new constant:

$$ROTR(a,n) = ((a >> n) \vee (a << (32-n)))$$
$$H4'' = ROTR(H4', 30)$$

Then, we may exit after $Round(75)$ if $E \neq H4''$. This removes a further 20 additions, 10 rotates and 30 assignments for an incorrect candidate.

4.1.2 HMAC

We apply three changes to the structure of the HMAC function.

First, because HMAC is based upon SHA1, our previous observations still hold, but there is a computational drawback to the HMAC construction, requiring modification of the Merkle-Damgård block-computational optimisation – we must execute the underlying SHA1 round-stage computation 4 times, rather than one. The use of the *ipad* to pad the key to the block-size (64 bytes), then concatenate this with the message requires 2 SHA1 round computations as the message length will now require execution of 2× 512-bit message blocks. Then the *opad* concatenated with the resulting 20 byte output again requires a further 2 round computations. Fig. 5 shows the necessary round-stage computations.

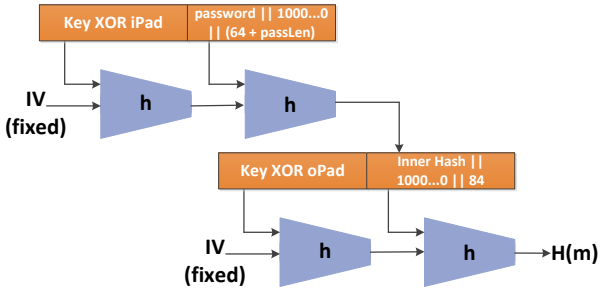


Figure 5: Modified Merkle-Damgård HMAC Block Construction

The primary cryptographic optimisation takes advantage of the fact that the key is known to be the same in a given cracking run for all password candidates. This allows us to pre-compute 2 of the SHA1 round computations on the host and transfer the 2 160-bit resultant hashes to the GPU global memory. This reduces the required round computations by 50% to 2. Figure 6 demonstrates this optimisation over figure 5, showing the pre-computed values as $oPadH$ and $iPadH$.

The two further changes to the algorithm structure are based upon the predictability of the input data that we pass to the function, removing 2 comparison operations. If $|K| \geq |H|$, then $K = H(K)$ which will never be the case in many of the considered implementations, thus is often

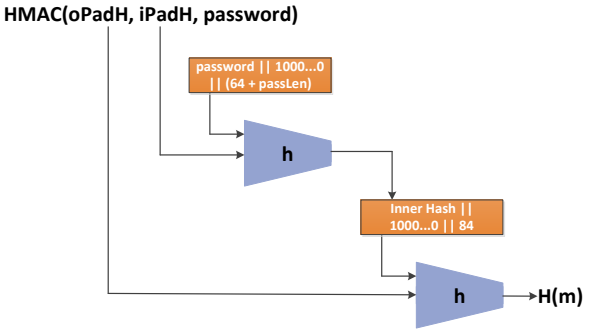


Figure 6: Merkle-Damgård HMAC Block Construction Optimisation

redundant. Additionally, if $|K| < |B|$, it must be 0 length padded, thus if the input key size is below $|K| = 64$ we may proceed directly to the padding [19].

4.1.3 PBKDF2

Due to the execution of the function U_{rc} 1000 times (as a specified minimum), the largest gains were achieved as a result of a design-flaw in the construction of the PBKDF2 primitive. This flaw allows pre-computation of the *iPad* and *oPad* values for the HMAC function, for all iterations. The round computations U_{rc} invoked within $F(P,S,c,i)$ key the HMAC function using the password and the message becomes the salt concatenated with the block ID for the first execution, chaining input for subsequent messages. The result of this is identical results of $(K \oplus opad)$ and $(K \oplus ipad)$ for all iterations, for all blocks T_i. This means that a single KDF block T_i requires only $2 + (2 \cdot c)$ SHA1 block computations, rather than $2(2 \cdot c)$ [19]. Meaning the PBKDF2 is susceptible to the same observation as that of the HMAC.

The second core optimisation is an early exit that targets the key-stretching. If $dkLen > |H|$, then multiple rounds of computation of T are unnecessary for an attacker. We need only compute the first block T_1, if this does not match the target hash, then the remaining computation is unnecessary. In the Microsoft .NET framework's implementation of PBKDF2, $dkLen = 256$ therefore we save 50% of the computational complexity when attempting an incorrect password candidate. Only if T_1 matches the equivalent portion of the target, is it necessary to compute T_2 and so on, until block T_{len}. For an unsuccessful candidate, we are cutting the workload considerably, by reducing the execution path to a single block, T_1. In an implementation requiring the computation of two blocks T_1 and T_2, where $c = 1000$, this reduces our workload from 8000 to 4000 SHA1 block computations. Combining the two aforementioned optimisations decreases this to just 2002 SHA1 blocks, for a

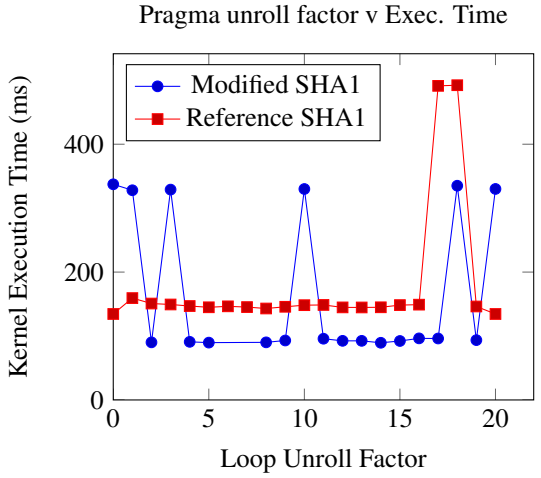
74.98% reduction in SHA1 round stages.

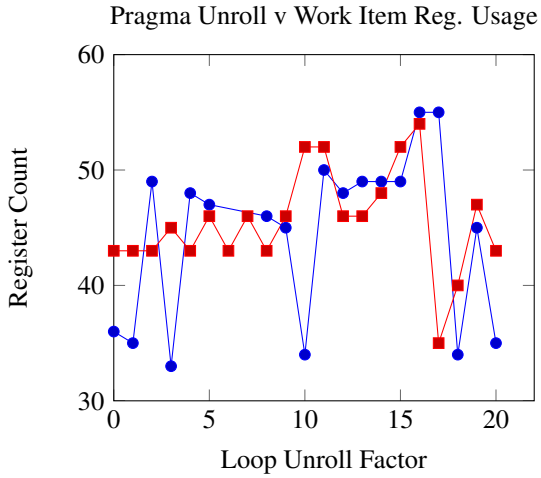
4.2 Kernel Optimisations

4.2.1 Unrolling / inlining

Comparison and branching operations are typically slow to process on a GPU, hence loop unrolling and inlining techniques can result in large speed-gains where a lot of operations exist within loop constructs. Unrolling and inlining should be carefully balanced against the increase in the size of the application binary as unroll factors affect occupancy levels.

We applied the OpenCL compiler loop unrolling directives ('#pragma unroll') with varying factors to test the compiler-generated instruction execution performance, on both a naïve implementation of the SHA1 hashing algorithm as well as a version including the removal of the Merkle-Damgård block construction. The unrolling was performed on the 80 $Round(t)$ stages.





When the loop has been unrolled too much, registers are spilled, increasing memory traffic and therefore kernel execution time. In the modified version, for the unroll factor of 10, there is $\sim$ 16% stall in the GPU write unit, resulting in 11.72% ALU usage, whereas the unroll factor of 11 has a 0% write stall for an ALU usage of 54.47%. All of the benchmarked kernels had an occupancy rate of 12.5% and no memory usage optimisation. The modified kernel took 337.31ms to complete one execution, processing 2^{22} password candidates. When an unroll factor of 18 was applied, there was just 2ms improvement in contrast to an unroll factor of 5 or 14, which gave an improvement of 248ms - a 73% reduction. Due to the low kernel occupancy rate of 12.5%, there is insufficient CU resource usage for effective global memory access latency hiding, which is partly responsible for the large jump in execution time, with a higher occupancy rate this figure would at least in part, be masked. These results show that the automatic loop unrolling performed by the AMD OpenCL compiler is not optimal.

As a result of the sporadic results produced by the compiler, we found it necessary to combine both the application of inline expansion, in conjunction with manual loop unrolling to realise optimal kernel performance.

```
1  // ROTATE_LEFT adapted from SHA1CircularShift
2  // as defined in IETF RFC3174
3  #define R2_S_BOX(A, B, C, D, E, W, tmp)      \
4  {                                             \
5      tmp = (ROTATE_LEFT(A,5)                   \
6          + (B ^ C ^ D) + E + W + K1);           \
7  }
8  #define SHIFT(tmp, A, B, C, D, E)             \
9  {                                             \
10     E = D;                                    \
11     D = C;                                    \
12     C = ROTATE_LEFT(B,30);                    \
13     B = A;                                    \
14     A = tmp;                                  \
15 }
16 // W(t) defined as a cyclic queue over W, as
17 // per NIST FIPS 180-4 section 6.1.3
18 #define R2_SHIFT(A, B, C, D, E, W, temp)     \
19 {                                             \
20     W(t);                                     \
21     R2_S_BOX(A, B, C, D, E, W, temp);         \
22     SHIFT(temp, A, B, C, D, E);               \
23 }
24 // R2 Manually unrolled:
25 #define R2()                                  \
26 {                                             \
27     R2_SHIFT(A, B, C, D, E, W[20], temp);     \
28     R2_SHIFT(A, B, C, D, E, W[21], temp);     \
29     R2_SHIFT(A, B, C, D, E, W[22], temp);     \
30     ...                                       \
31     R2_SHIFT(A, B, C, D, E, W[39], temp);     \
32 }
```

4.2.2 Bus Data Transfers

In order to observe the effect of bus transfer speeds on our overall application hash throughput rates, we implemented functionality to transfer all computed hashes back to the host to detect a hash collision, rather than computing this collision on the GPU.

We found that when computing a collision on the GPU, SHA1 spends $\sim$ 60% of its execution time pro-

cessing memory requests, in contrast to its host equivalent, which spends $\sim 82\%$. In our implementation, due to the large workload increase, PBKDF2 utilises 0.1% of processing time for memory transfers, whereas the host version utilises 0.19%, approximately double. It is worth noting that for PBKDF2, the key-stretching early exit optimisation cannot be applied if the host computes the collision, without multiple kernel calls and bus transfers.

In all cases, the small kernel processing overhead in GPU collision detection saves much memory transfer time. In SHA1 the reduction is 30.90%, HMAC is 24.12% and PBKDF2 is 31.03%.

Figure 7 demonstrates the effect of these transfer schemes on various size password blocks.

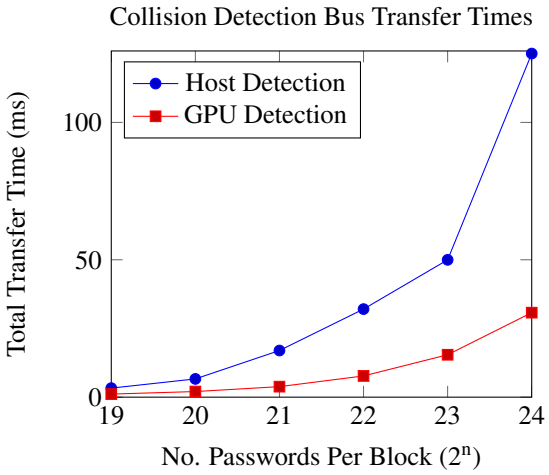


Figure 7: Collision Detection Bus Transfer Times, HD6870

4.2.3 Occupancy/Latency Hiding

Kernel occupancy rates are important for effective masking of memory access latencies. Where register usage of an algorithm is too large to allow for sufficient in-flight wavefronts, two key areas need to be considered.

The first is to look at reducing the volume of direct memory accesses, with a view to reducing the total length of any stalls introduced. This can be achieved by moving data storage onto the LDS memory banks to allow for a shorter memory access latency, especially where frequent reads and writes occur.

The second method is to look to reduce the kernel processing complexity through a reduction in register usage. This method can be harder to assess exactly where optimisations should be made, however we have demonstrated the effect of compiler-driven unrolling techniques on the active kernel register count, which should make clear that optimal levels of unrolling and inlining have a role to play in this, as does the branching complexity of a given piece of code – code divergence should be avoided where possible.

Inevitably, the more complex a kernel becomes, the higher the register count will be (the complexity depends on a number of factors, including divergant code paths, cross-WG memory barriers and multiple function calls). In our reference implementation of PBKDF2 too many device registers were consumed for the kernel to actually compile for the device architecture. As a result of this, it is necessary for some memory optimisations in order to even get a complex algorithm to run.

A high ALU utilisation ($\geq 90\%$) and low memory stall figures are indicative of sufficient latency hiding, in which case further reduction in register count is unlikely to provide much performance increase when compared to instruction count reductions. Conversely when the ALU utilisation is low with memory stalls and register count usages high, increases in occupancy levels will often map directly to output speed (e.g. 12.5% occupancy increase to 25% represents a $\sim 100\%$ increase).

4.2.4 Memory Access Coalescing

If data storage is pushed onto LDS banks to reduce CU register memory usage, then ensuring that a co-operative coalesced read takes place, rather than reading across the dataset as a whole will reduce the length of any stalls introduced. Each Work-Group has its own allocation of LDS storage, thus to coalesce reading of data from LDS, we must ensure that each read is the size of a given WG multiplied by the number of in-flight wavefronts per execution within that WG.

Reading data from LDS is significantly less expensive than from global memory. We efficiently coalesce access across each WG, ensuring that global memory is read only once per lock-step execution, any subsequent data access penalties will be substantially lower. This ensures that a single allocation of LDS storage is made per WG, allowing lock-step execution to index into the local storage bank. This decreases our processing times and our required kernel register count. This can be achieved as follows:

```
1  //Kernel Instance Global GPU Mem IO Mapping:
2  int id = get_global_id(0);
3  int localId = get_local_id(0);
4  int inputIndexStart = id * passwordLen;
5  //Coalesce read across Wavefront:
6  __local uchar wavefrontInput[passwordLen *
       256]; //Read block of 4 * 64 input values
7  __local uchar* threadRead = (__local uchar*)&
       wavefrontInput[localId * passwordLen];
8  //Cooperative coalesced read:
9  #pragma unroll
10 for (i = 0; i < passwordLen; i++)
11 {
12     threadRead[i] = in[inputIndexStart + i];
13 }
```

4.2.5 Instruction Packing

GPU register files have 32-bit elements, so it is important to ensure that when performing bitwise operations that the data representation being used effectively packs into the registers. Initially, we were performing the XOR operations of the PBKDF2 F function on 8-bit uchar values, however manual inspection of the produced ISA instructions revealed that the compiler wasn't efficiently packing our bitwise operations and was introducing 4x the required number of operations. Manually packing the data types into 32-bit uint values, rather than performing the operations on 8-bit data types effectively reduced the instruction count and gave us approximately a 12% increase.

4.2.6 Effect of Work-Group Sizes

We also investigated the effect of various work-group sizes on kernel throughput times. As shown in Fig. 8, we achieved the most optimal results with a single wavefront per work-group (WG = 64). This demonstrates an additional overhead that has a negative performance impact once the given work-group size is already large-enough to mask any memory access latencies experienced on the device. This overhead is attributable to the device wavefront context switching and can be seen by looking at the kernel occupancy rates – a single wavefront has an 18.5% kernel occupancy rate compared to 2 or 4 waves per work-group (WG = 128 or 256), which has 12.5%. The work-item register usage was 39 in all cases.

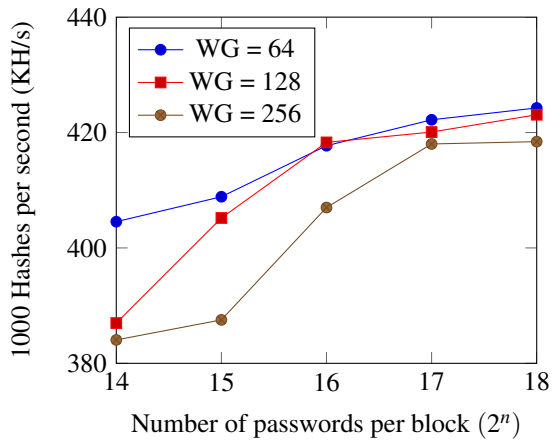


Figure 8: Effect of Work-Group Size and Password Block Size on Throughput (HD6870)

5 Results

Results were obtained on the following mid-range consumer hardware: ATI HD6870 GPU device with 1GB GDDR5 and a 3.7GHz AMD Phenom II 560BE. All benchmarks were run on the Windows 7 operating system and all figures we present treat 1KH/s as 1,000 hash operations per second and 1 MH/s as 1,000,000. Our core focus here is not the password generation but the kernel execution times, thus all figures quoted are independent of any memory transfers. For our results, $c = 1,000$ and $l = 2$.

We present a total of 11 optimisations (7 algorithmic and 4 kernel); as summarised in figure 9, items (1) – (7) represent the algorithm optimisations and items (8) – (11) show the kernel optimisations. The individual application of various optimisations is intended to indicate the level of contribution each make to the final result, however at any given stage, a mix of various kernel and algorithmic optimisations would produce much more impressive figures.

We noted some strange results when considering some of the versions of the algorithms that had no memory or kernel optimisations. Sometimes sporadic or incorrect results would be produced from otherwise correct kernel code and manual debugging would yield the correct values, but directly executed code would not. We tracked this down to a compiler bug. Sometimes a particular block of code will return an incorrect value (always within a loop), yet forcing the compiler to not perform auto-optimisation by directing an unroll factor of 1 returns the correct result and the kernel execution time increases. This supports the explanation that sometimes, the AMD OpenCL compiler produces invalid results when performing automatic loop-unrolling code optimisations. As such, results where there is a negative performance impact due to algorithm optimisations are attributable to compiler bugs that prevented presentation of comparable figures.

Although the percentage increase in speed looks far more impressive in favour of the kernel optimisations (Fig. 9) the largest impact on the overall performance in PBKDF2 is actually in the algorithmic optimisations. The number of rounds of SHA1 transforms is decreased by 74.98% as a direct result of the algorithm optimisations discussed in 4.1.3 (ignoring the HMAC and SHA1 optimisations altogether). The kernel optimisations gave $\sim 24.71\times$ improvement over the naïve solution, that is to say $24.71\times$ the remaining 25.02% of operations, which is far lower than it would be without optimisations (8) and (9) in Fig. 9. This demonstrates the massive increase due to the PBKDF2 flaws.

Hash	SHA1	HMAC-SHA1	PBKDF2
Unoptimised	31.47 MH/s	1.35 MH/s	903.71 H/s
(1) 1 Block Only	19.33 MH/s	–	–
(2) Cyclic Storage	28.32 MH/s	–	–
(1) + **(2)**	58.34 MH/s	–	–
(3) HMAC Merkle-Damgård	–	1.77 MH/s	–
(4) HMAC Comparison Removal	–	1.14 MH/s	–
(3) + **(4)**	–	1.77 MH/s	–
(3) + **(5)** HMAC Storage Reuse	–	6.84 MH/s	–
(2) + **(3)** + **(4)** + **(5)**	–	8.01 MH/s	2.13 KH/s
(6) *iPad* & *oPad* pre-computation (with **(2-5)**)	–	–	5.01 KH/s
(7) Key Stretching Optimisation	–	–	4.27 KH/s
(2-7)	–	–	16.41 KH/s
(8) Unrolling / Inlining (with **(1-7)**)	752.93 MH/s	60.50 MH/s	367.91 KH/s
(9) Memory Optimisations (with **(1-7)**)	57.70 MH/s	9.89 MH/s	16.40 KH/s
Fully Optimised (without [19])	776.01 MH/s	132.16 MH/s	370.05 KH/s
(10) Fully Optimised (with [19])	**794.60 MH/s**	**395.21 MH/s**	–
(11) Fully Optimised (with integer packing optimisations)	–	–	**424.78 KH/s**

Figure 9: Optimisation results on GPU

6 Comparison With Prior Art

6.1 Prior Art

A CUDA accelerated PBKDF2-HMAC-SHA1 implementation, with $c = 1,000$ in [11], reports $\sim$ 65 KH/s throughput on an NVIDA GTX480. They use reference cryptographic code, adapted to the CUDA programming model. Only a single block of T_i is calculated as $l = 1$.

A CUDA accelerated PBKDF2-HMAC-SHA512 implementation, with $c = 1,000$ and $l = 1$ is presented in [9]. This paper reports a throughput figure of 54,874 H/s in their demonstration tool on 4$\times$ NVIDIA Tesla C2070. They implement the HMAC *oPad* and *iPad* precomputation and the PBKDF2 extension of this ((3) and (6) in Fig. 9, respectively). This paper focuses on the contrast of the GPU performance against that of a RIVYERA FPGA cluster.

A CUDA implementation of PBKDF2-HMAC-SHA1 is reported in [16]. A SHA1 throughput figure of 229.9 MH/s is presented and a PBKDF2 figure of 109.04 KH/s, with $c = 2,000$ and though it is not explicitly mentioned, it is hinted that $l = 1$ on a single NVIDIA GTX280. For direct comparison to the implementations where $c = 1,000$, we can scale this figure by 2$\times$ for 218.8 KH/s. Kernel password input is faked in order to test throughput rates - a single block start value is transferred to the GPU, each kernel adds its index id to this start value, which is assumed to be padded to RFC3174 [10], before it is hashed.

A further CUDA implementation of PBKDF2-HMAC-SHA1 is presented in [15]. Utilising 2$\times$ dual-GPU GTX295 cards, the paper focuses on the application of context-free grammars to reduce the password search space and presents a GPU throughput of just 5,011 H/s.

Two primary off-the-shelf password crackers are available on the market at present: John the Ripper [3] and oclHashcat [17], with the latter being the state-of-the-art PBKDF2. When we started work on our implementation, the latest development version of John the Ripper was 1.7.9-jumbo-6, which did not support PBKDF2. At the time of publication, 1.8.0-jumbo-1 has added this support. Our effort was entirely independent from and completed ahead of John the Ripper's OpenCL PBKDF2-HMAC-SHA1 implementation.

6.2 Comparison With Our Work

A direct comparison of the quality of optimisations implemented in each of the considered works was not possible as we do not have access to source code or the GPUs utilised. Figure 10 presents a relative comparison of the cost of each of the GPUs utilised, compared to our HD6870 as well as their relative performance benchmarks when applied to the Bitcoin mining algorithm (as this is a thoroughly researched area) [1]. It is worth mentioning that based on current market offerings, ATI GPU's generally provide a better cost to performance ratio for hash computation than NVIDIA devices.

Research Team	Inner Hash Function	GPU	BitCoin Mining Speed v HD6870	Second-hand cost v HD6870	PBKDF2 Throughput (KH/s)
[11]	SHA1	GTX480	0.6×	1.5×	65.00
[9]	SHA512	4 Tesla C2070	3.2×	60×	54.87
[16]	SHA1	GTX280	0.28×	0.75×	218.8
[15]	SHA1	2 GTX295	1.01×	2×	5.01
Us	SHA1	HD6870	1×	1×	424.78
oclHashcat [17]	SHA1	HD6870	1×	1×	391.9

Figure 10: Performance Comparison with HD6870 (c = 1000)

The only other source we have found that describes a similar architecture that we discuss in Section 3 is [16], which reports their highest PBKDF2-HMAC-SHA1 speeds at 218.8 KH/s, though they do not consider an implementation that makes use of key stretching. It is as a result of the HMAC *ipad* and *opad* pre-computation optimisaions that we achieve a $\sim 48\%$ speed improvement. Had they also considered multiple T_i block computations, we would be able to present roughly double this improvement again. Additionally, we found that the password input data did not need to be faked. Due to the run-time length of the PBKDF2 kernel, there is ample time to generate password data blocks on the CPU, such that we do not introduce memory stalls. The bus memory transfer time to the GPU is greater in our implementation, however this proves to add a negligible overhead – processing a block of 2^{19} passwords leads to an equivalent decrease of just 90 hash operations per second. We also perform the message padding operations on the GPU, which is not done in [16] – removing this work artificially increases the hash rates they present by eliminating necessary work. Our work performs all necessary calculations and still presents a 50% – 75% improvement (dependant upon l).

An unpublished manuscript [11] attacked PBKDF2-HMAC-SHA1, and thus is directly comparable to ours. However, they achieved only a rate of 65 KH/s. Our re-implementing the cryptographic primitives from scratch, with a focus on improving, optimising and exploiting their designs, rather than re-using reference implementations as done by them, results in a massive speed gain of 5.56×. On the other hand, some possible optimisations were speculated in [11] without empirical backing. Our experience confirms some of their speculations, but debunks others. For example, their assertion that loop unrolling provides questionable improvements is not sound, we demonstrate in excess of 70%. Some speculations are sound, such as those on coalescence, buffer recycling and 32-bit word operations.

The figure of 54.87 KH/s achieved by [9] in attack of the PBKDF2 operating over a SHA512 is very impressive. However, the hardware used in their implementation costs over 60× ours, yet it only yields a factor of 3.2 when applied to Bitcoin mining. A realistic estimate of the added complexity of SHA512 compared to SHA1 is around 11.97× (this number was obtained through averaging the figures presented on 6 separate hardware configurations by oclHashcat [17]). If we factor our result by these differences (ignoring the hardware cost implications) our number for comparison is $424.43 \div 11.97 = \sim 35.46$ KH/s on a single GPU. If the cost is factored in, we believe their figure for direct comparison is less than 5% of our result (each GPU utilised is 15× more expensive than ours and yields ~ 13.72 KH/s). Therefore, our work could significantly boost both their acceleration and subsequently, their password cracking performance.

6.3 oclHashcat

OclHashcat considers both algorithmic and kernel optimisation. Cryptographic exploits in the underlying SHA1 were released in [18, 19] by the author, Jens Steube. Tested in benchmark mode on the HD6870, oclHashcat v1.37 (released August 2015) reports the SHA1 throughput as $\sim 1{,}462$ MH/s, HMAC-SHA1 $\sim$ 592.7 MH/s and PBKDF2-HMAC-SHA1 with $c = 1000$ $\sim$ 391.9 KH/s. Previous versions of oclHashcat, such as v1.31 did not have a directly comparable PBKDF2 implementation.

We tested our code unchanged, on ATI's current flagship GPU, the R9 290X, which is much improved over the older HD6870 for GPGPU computation. Figure 11 presents our results, along with oclHashcats on this new architecture (both tested with a single R9 GPU).

This yields interesting results. There is a significant improvement when our code is executed on the newer architecture: our HMAC implementation is just $\sim 6.14\%$ slower than oclHashcat. This is in contrast to the H6870 which was $\sim 33.32\%$ slower. Therefore, it becomes

	SHA1 (MH/s)	HMAC (MH/s)	PBKDF2 (KH/s)
Our Implementation	3,415.37	1,610.62	1,611.98
oclHashcat	4,142	1,715.9	1,451

Figure 11: Results on ATI R9 290X

apparent that the ISA code produced by the evergreen compiler for our kernel code is sub-optimal. Examining the disassembly of the HMAC kernel on both architectures shows a 5.25% decrease in the number of ISA instructions generated. It is worth mentioning that we noticed a discrepancy in the benchmark figures reported by oclHashcat and the speed reported by a brute force attack when executed on our system, which reported a figure of $\sim$ 1,485 MH/s, rather than 1,715.9 MH/s. Therefore in practice our kernel was observed to operate at a gain of 8.5% on the R9 290X.

The $\sim$ 17.54% difference in oclHashcat's SHA1 performance is attributable to the fact we chose not to implement an optimisation allegedly providing a 21.1% increase [18]. We favor instead the cyclic storage optimisation (2), as shown in figure 9. The two optimisations are mutually exclusive. Implementing [18] would have been incompatible with our password generation algorithm and is defeated by the construction of HMAC and PBKDF2. Therefore 3.56% is a realistic estimate of the improvement provided by optimisation (2). For SHA1, oclHashcat achieved $\sim 3,417$ MH/s in a brute force attack on our hardware.

Our PBKDF2 implementation was $\sim$ 8.4% faster than oclHashcat on the HD6870, but the R9 increases our lead to $\sim$ 11.09%. There is a much more pronounced difference on the newer architecture for SHA1 and HMAC, when compared to the HD6870 than there is for PBKDF2. This is either to do with the compilers inability to optimise the more complex code, or it is that oclHashcats implementation does not map equally well to the new hardware. It is our assertion that this difference is likely due to a mixture of both. Whilst PBKDF2 is composed of both SHA1 and HMAC constructs, this should make it clear that an optimal SHA1 does not necessarily lead to an optimal HMAC, and an optimal HMAC does not necessarily lead to an optimal PBKDF2.

7 Practical Application

We apply our PBKDF2 implementation to attack the Microsoft .NET role-provider security model, upon which Microsoft's flagship web-server framework, ASP.NET MVC, bases its password hashing. It was estimated that in 2016 about 15% of all websites in the world run on ASP.NET [4]. In this system, Microsoft uses a PBKDF2-HMAC-SHA1 with $c = 1000$ and $l = 2$. We chose it as our target because of its practical significance, and also because it is not directly targeted by any other hacking tools.

Our code has an execution time of 10.36 minutes per 1 billion password candidates, utilising our real throughput speed (including all bus memory transfers) of 1,608.86 KH/s on the R9 290X. Whilst this number is too low for a brute-force attack over a large key-space size such as 62^8, it is high enough to make a dictionary attack feasible.

The cumulative probability of cracking any single password within a database containing multiple password hashes, on attempt n, where the individual probability of the password being contained within our dictionary (ip) is $p(n) = 1-(1-ip)^n$.

An unverified source claims an 18.2% success rate with a dictionary containing 1.494 billion passwords on a leaked database of actual user passwords from the website, eHarmony. Both the dictionary file and the source database are available online [2], however due to the ethically questionable nature of verifying the findings, we did not attempt to do so. If this is a representative dataset, we have a high probability ($\sim$ 0.9) of finding a crack after trying just 10 or 11 passwords against the dictionary, which would take 2.58 - 2.83 hours. The cumulative probablity of finding a crack and the time required to do so is presented in fig. 12.

This means it is within the reach of a home desktop or laptop computer to crack some 8 ASP.NET passwords per day. This is a hugely worrying result for the security of this framework.

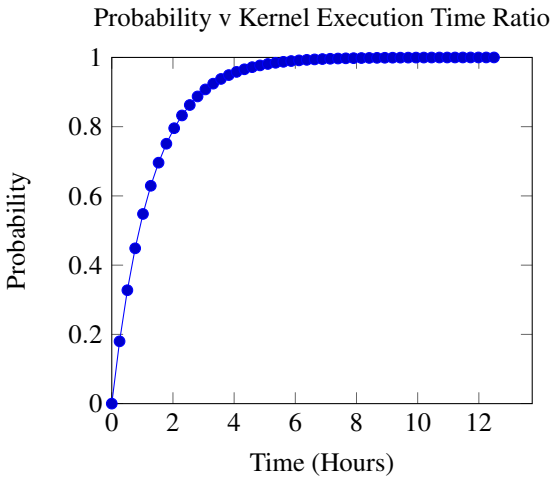


Figure 12: Cumulative Probablity of a Single Crack

As a side note, we notice that our attack will work well with WPA2 for the following reasons. WPA2 is based upon the same HMAC-SHA1 primitive with $dkLen =$

256 and $c = 4,096$. Therefore, 2 rounds of 4,096 iterations, each requiring 4 SHA1 stages as part of the HMAC construction results in 32,768 SHA1 round-stage computations per candidate password. Our attack will reduce this number to just 8,194, 4 times faster. We did not implement a practical attack on WPA2, as the process of extracting the PBKDF2 hash and salt (SSID) from an active WiFi connection is a well-known one, and our work is directly applicable to this usage, with minor modification. A recent paper [14] at WOOT'15 describes the WiFi de-authentication process, whereby an attacker can inject a packet into an active connection, forcing the subsequent client re-authentication. A packet capture of this re-authentication (obtainable from a tool such as the open-sourced wireshark [5]) allows the retrieval of the PBKDF2 hash. The Linux distributions Backtrack and Kali include open-source tools (including aireplay-ng), that easily allow such an attack to be mounted. We estimate an approximate throughput of 393 KH/s on the R9 290X, using our attack.

8 Conclusions

We have investigated accelerating PBKDF2 by considering both algorithmic optimisations in the underlying primitives and OpenCL kernel code optimisations. The design of our acceleration attack has followed a best-effort approach, and we have achieved the state-of-the-art acceleration of both the HMAC and PBKDF2 primitives.

We empirically evaluated 11 factors that contribute to acceleration, measuring the contribution of each and contrasting the performance of both optimisation categories. In doing so, we draw a surprising conclusion: in contrast to the common view that GPU acceleration is primarily driven by massively-parallel hardware architectures, we have demonstrated that a larger contribution is made through cryptanalytic optimisations on the underlying primitives.

Another new and counter-intuitive insight we have learned is the following. Whilst PBKDF2 is composed of both SHA1 and HMAC constructs, an optimal SHA1 acceleration does not necessarily lead to an optimal HMAC acceleration, and an optimal HMAC acceleration does not necessarily lead to an optimal PBKDF2 acceleration.

We have also revealed some insights on the black-box of oclHashcat. First, our empirical analysis suggests that it is algorithmic optimisations that contribute the most to oclHashcat's ability to drastically outperform competitors. Cryptanalysis efforts have led to significant algorithmic optimisations, which when combined with parallelism achieved previously the highest acceleration. Second, our best-effort approach utilises both our own innovations and all cryptanalytic exploits known in the hacker community. As we are able to match and even exceed oclHashcats performance on the latest generation of hardware, we are confident to conclude that the chance of any further significant cryptographic exploits hidden in oclHashcat's implementation is very low.

Moreover, we have demonstrated and discussed the real threat that our acceleration attacks pose to actively deployed popular systems such as Microsoft .NET Framework and WiFi Protected Access (WPA2). Awareness of these issues needs to be increased, given the large number of systems that implement PBKDF2 we have identified. In this respect, we have the following recommendations. We have shown that extending the length of the derived key *dkLen* beyond the length of the underlying hash function *hLen* does not add to the security of the function and in fact, places a legitimate user of the system at a computational disadvantage against an attacker – each additional block T_i beyond the first, has the effect of reducing the workload of the attacker by a factor of two, or $log_2(l)$. Where larger derived keys are needed, use SHA256 or SHA512 as the underlying hash function.

Additionally, the PBKDF2 HMAC construction contains a design flaw allowing *ipad* and *opad* precomputation, further reducing the work required of an attacker from $2(2 \cdot c)$ SHA1 blocks to $2 + (2 \cdot c)$ blocks. This flaw is a result of incorrect keying of the underlying HMAC function – if the functional composition was such that the message was the password and $S||int(i)$ the key, we would be unable to exploit this. Though it would still prove possible to pre-compute the first block of both the inner and outer HMAC applications, which would be the same for every password candidate $\big((salt||int(1)) \oplus ipad\big)$ and $\big((salt||int(1)) \oplus opad\big)$, providing a gain of $(2 \cdot l)$ minus the number of candidates overall. SHA2 variants of PBKDF2 are still susceptible to this flaw. As such, our recommendation follows that the PBKDF2 cryptographic standard in PKCS#5 be updated to take advantage of the construction outlined by Yao and Yin [21], whereby the construction of $F(P,S,c,i)$ instead uses a chained construction over $H(p||s||c)$.

Alternatively, library implementations of PBKDF2 should be updated to take advantage of the same optimisations and the iteration count c be increased by $2(2 \cdot c) - 2 + (2 \cdot c)$ to provide the same, intended level of security. As such, the specified minimum iteration count of 1,000 adjusted for the aforementioned optimisations, needs to increase to 1,998, ignoring hardware advances altogether.

To better protect against GPU-accelerated attacks, researchers and developers can resort to alternative password-based key derivation functions based on memory-hard functions, which have been an active area of research in recent years.

Acknowledgment

We thank Alexander Lyashevsky at AMD Research, San Francisco, CA, and Szűcs István of Eötvös Loránd University, Hungary for their help. We also thank Vashek Matyáš and Milan Broz of Masaryk University, Czech Republic for their helpful comments.

References

[1] Bitcoin Mining Hardware Comparison, https://en.bitcoin.it/wiki/Mining_hardware_comparison

[2] Cracking eharmony's unsalted hashes with crackstation, https://defuse.ca/blog/cracking-eharmonys-unsalted-hashes-with-crackstation

[3] John The Ripper Password Cracker, http://www.openwall.com/john/

[4] Usage statistics and market share of asp.net for websites, http://w3techs.com/technologies/details/pl-aspnet/all/all

[5] Wireshark network protocol analyzer, https://www.wireshark.org

[6] The Keyed-Hash Message Authentication Code (HMAC). FIPS PUB 198-1 (Jul 2008)

[7] Secure Hash Standard (SHS). FIPS PUB 180-4 (Mar 2012)

[8] Czajkowski, T., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D.: From OpenCL to High-Performance Hardware on FPGAs. In: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on. pp. 531–534 (Aug 2012)

[9] Dürmuth, M., Güneysu, T., Kasper, M., Paar, C., Yalcin, T., Zimmermann, R.: Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In: ESORICS 2012, pp. 716–733 (2012)

[10] Eastlake, D., Jones, P.: US Secure Hash Algorithm 1. RFC 3174 (2001)

[11] Frederiksen, T.K.: Using CUDA for Exhaustive Password Recovery, Unpublished Draft, http://daimi.au.dk/~jot2re/cuda/resources/report.pdf

[12] Kaliski, B.: PKCS #5: Password-Based Cryptography Specification, 2.0. RFC 2898 (2000)

[13] Krawczyk, H., Bellare, M., Canetti, R.: RFC 2104: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Feb 1997)

[14] Lorente, E.N., Meijer, C., Verdult, R.: Scrutinizing wpa2 password generating algorithms in wireless routers. In: 9th USENIX Workshop on Offensive Technologies (WOOT 15) (2015)

[15] Phong, P.H., Dung, P.D., Tan, D.N., Duc, N.H., Thuy, N.T.: Password Recovery for Encrypted ZIP Archives Using GPUs. In: Proceedings of the 2010 Symposium on Information and Communication Technology. pp. 28–33. ACM (2010)

[16] Schober, M.: Efficient Password and Key Recovery Using Graphic Cards. Master's thesis, Ruhr-Universität Bochum (2010)

[17] Steube, J.: Hashcat Advanced Password Recovery, http://hashcat.net/oclhashcat/

[18] Steube, J.: Exploiting a SHA1 Weakness in Password Cracking (2012), http://passwords12.at.ifi.uio.no/Jens_Steube/Jens_Steube_Passwords12.pdf

[19] Steube, J.: Optimising Computation of Hash-Algorithms as an Attacker (2013), https://hashcat.net/events/p13/js-ocohaaaa.pdf

[20] Turan, M.S., Barker, E., Burr, W., Chen, L.: Recommendation for Password-Based Key Derivation. FIPS SP 800-132 (Dec 2010)

[21] Yao, F., Yin, L.: Design and Analysis of Password-Based Key Derivation Functions. In: IEEE Transactions on Information Theory, vol. 51, pp. 3292–3297 (2005)