Acceleration Attacks on PBKDF2

Or, what is inside the black-box of oclHashcat?

Andrew Ruddick, UK

Dr. Jeff Yan, Lancaster University, UK

andrew.ruddick@hotmail.co.uk, jeff.yan@lancaster.ac.uk

What is PBKDF2?

- Password Based Key Derivation Function v2 (PBKDF2)
    - Standardised as NIST FIPS SP 800-132 and IETF RFC 2898
- Key-stretching Algorithm
- Based on an underlying hash-function, e.g. SHA-1x, SHA-2x, or MD-x
    - We look at PBKDF2-HMAC-SHA1, the most popular implementation
- Used by Microsoft, Apple, Cisco, Google and WiFi
    - WPA/ WPA2, Microsoft .NET, Microsoft Windows Data Protection API (DPAPI), Apple OS X OS User Passwords, Apple iOS passcodes / passwords, Cisco IOS Type 4 passwords, Android Full Disk Encryption (v3+), TrueCrypt … and many many more

Our Contribution

1. What are the limits of acceleration on cheap, commodity GPUs?
    - In pushing the limits, what new deep insights can be learned?
2. What are the relative contributions of various optimisations on acceleration?
    - Algorithmic Optimisation
    - OpenCL Kernel Code Optimisation
3. Why does oclHashcat outperform competitors?
    - Do they exploit hidden cryptographic vulnerabilities?
    - Can we improve its acceleration?
4. A practical attack on Microsoft's .NET Framework
    - We will release our code: https://github.com/OpenCL-Andrew/.NETCracker/

PBKDF2 Construction

$$\boldsymbol{HMAC}(K, M) = H\Big((K \oplus opad) \,||\, H\big((K \oplus ipad) \,||\, M\big)\Big) \tag{1}$$

$$\boldsymbol{PBKDF2}(Pass, Salt, count, dkLen) = (T_1 \,||\, T_2 \,||\, \ldots \,||\, T_{l\text{<Can be partial block>}}) \tag{2}$$

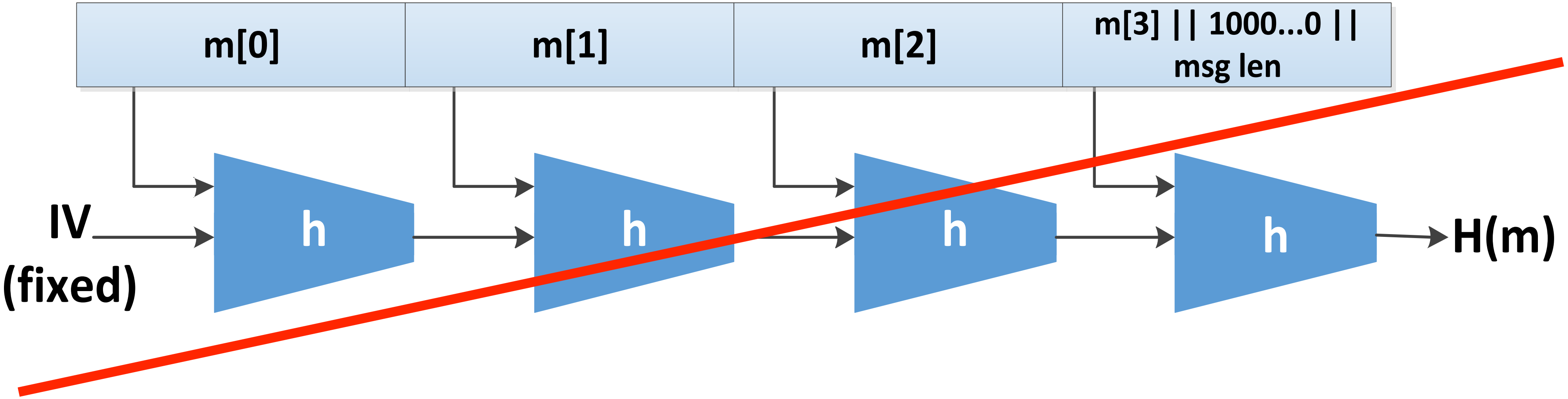
$$T_i = F\big(Pass, Salt, count, i\big) = \big(U_1 \oplus U_2 \oplus \ldots U_{count}\big) \tag{3}$$

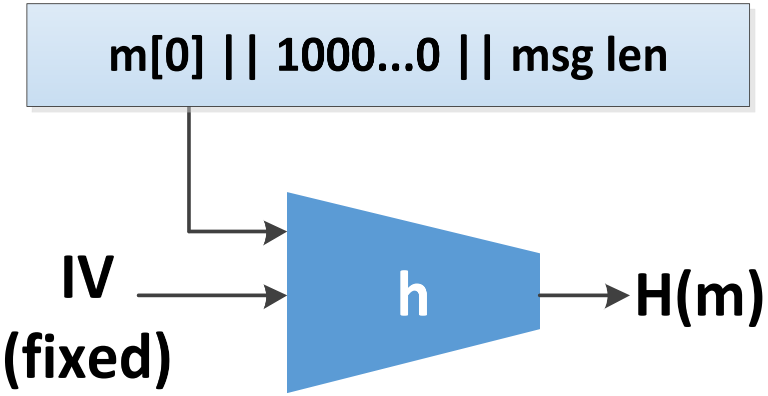
$$U_{rc} = \begin{cases} U_1 = HMAC(Pass, Salt \,||\, int(i)) & \text{1st iteration} \\ U_2 = HMAC(Pass, U_1) & \text{2nd iteration} \\ \vdots & \vdots \\ U_c = HMAC(Pass, U_{count-1}) & \text{Final Iteration} \end{cases} \tag{4}$$

PBKDF2 Optimisations – Cryptanalytic

- Merkle-Damgård optimisations
- PBKDF2 key stretching
- Zero-based optimisations
- Cyclic storage optimisations
- S-Box optimisations (not discussed in paper)

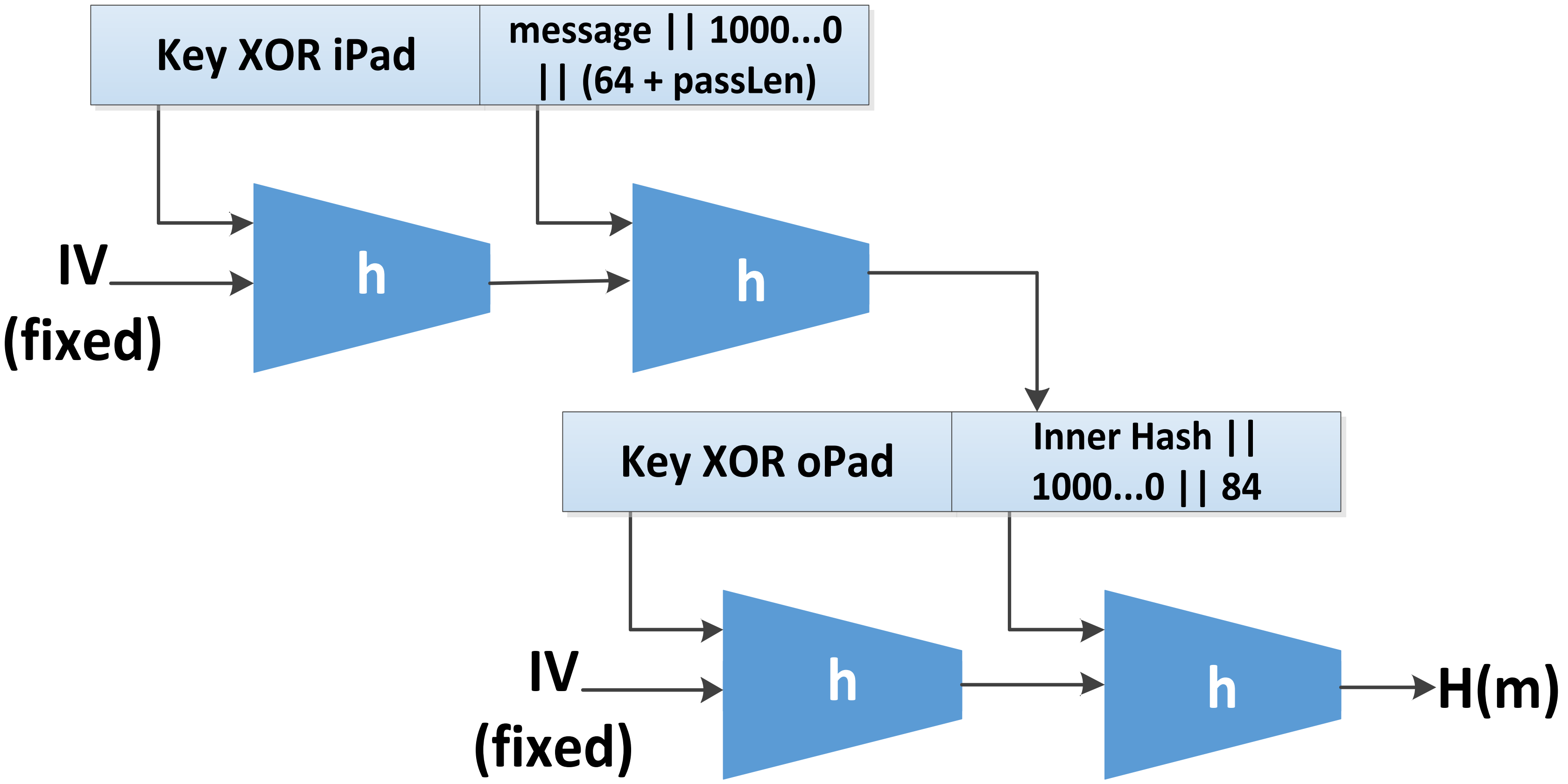
Optimisations – Merkle-Damgård (SHA1)





Optimisations – Merkle-Damgård (HMAC)

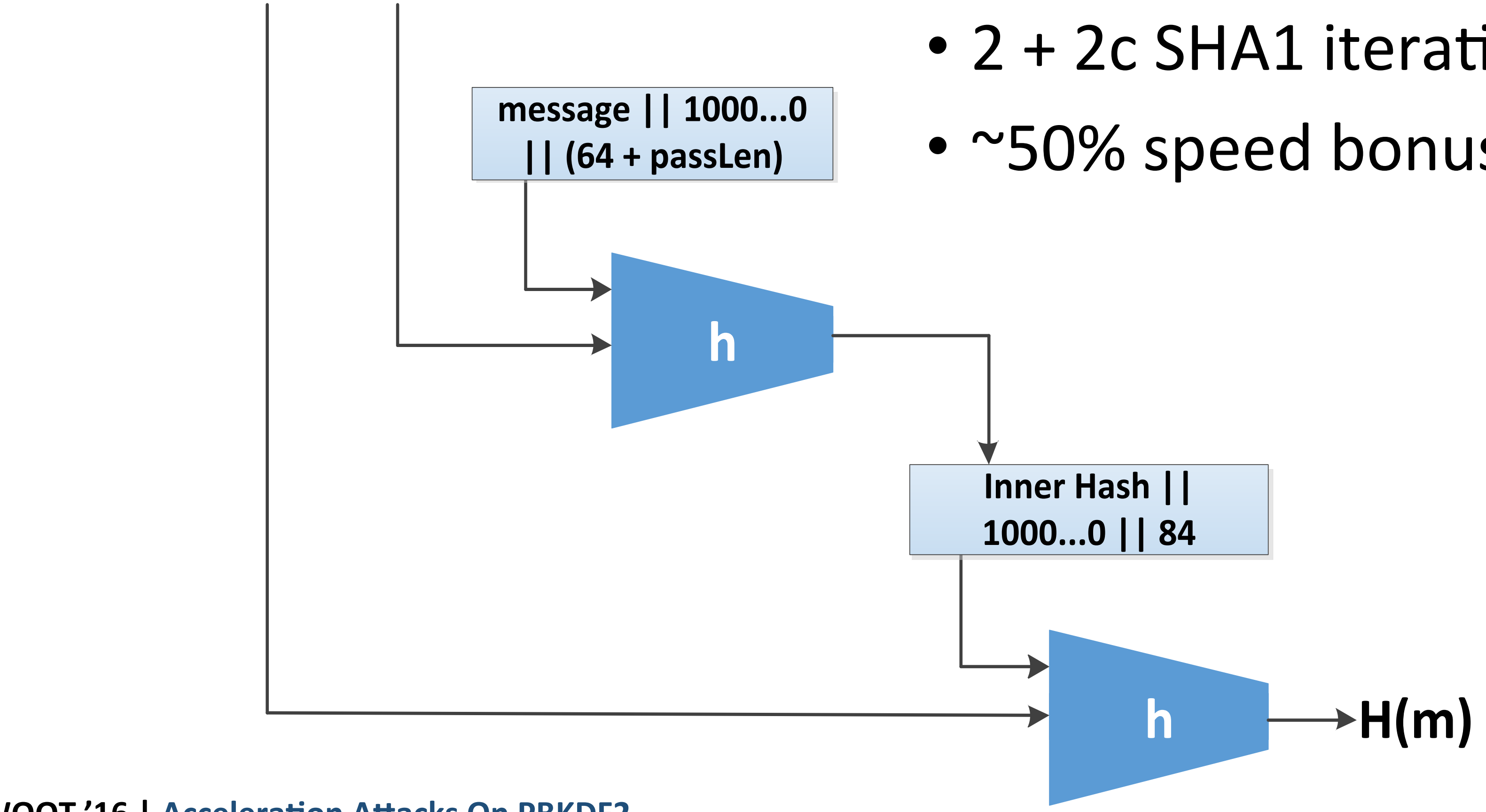
$$HMAC(K,M) = H\Big((K \oplus opad) \,||\, H\big((K \oplus ipad) \,||\, M\big)\Big)$$



Optimisations – Merkle-Damgård (PBKDF2)

- $(pass \oplus opad),\ (pass \oplus ipad)$ known to be the same for all iterations

HMAC(oPadH, iPadH, message)



- 2 + 2c SHA1 iterations, instead of 4c
- ~50% speed bonus for an attacker

Optimisations – Key Stretching (PBKDF2)

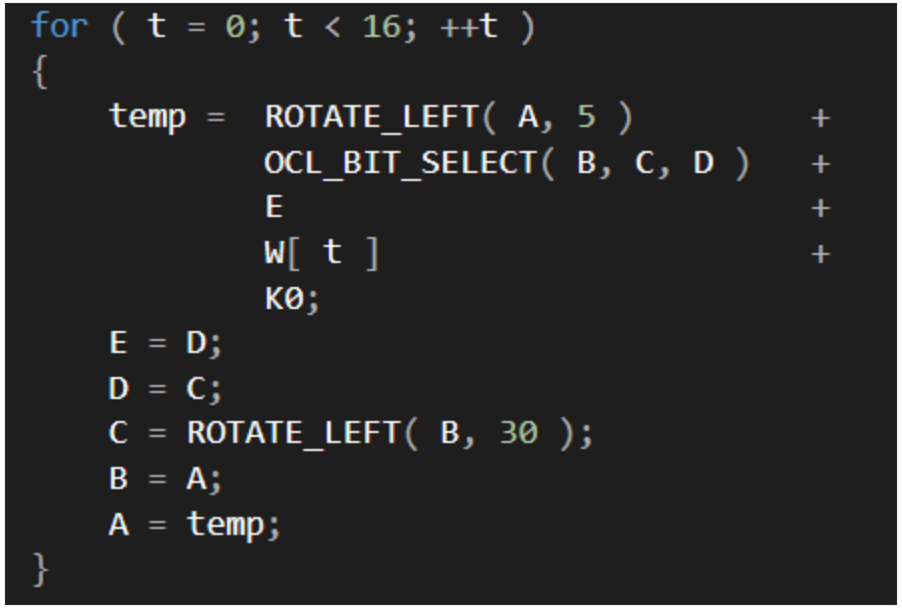
- An early exit optimisation targeting key stretching:

$$\boldsymbol{PBKDF2}(Pass, Salt, count, dkLen) = (T_1 \,||\, T_2 \,||\, \ldots \,||\, T_{l<\text{Can be partial block}>})$$

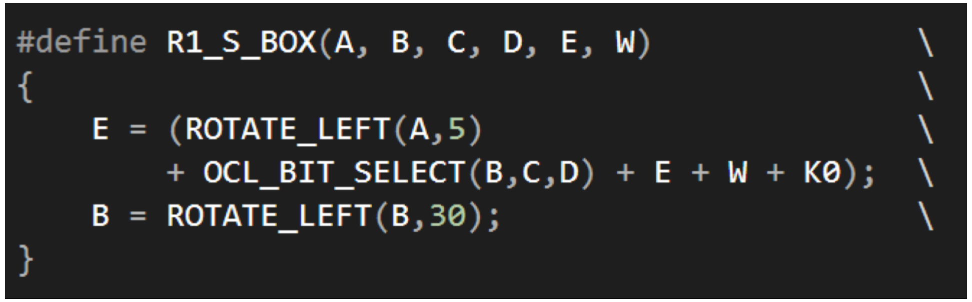
- If multiple iterations required, just calculate the first
  - Match? Probably a crack, check next block (or don’t. SHA1 = 2^{160} entropy).
  - No match? Early exit.
- A ***further*** 50% bonus for an attacker, in an implementation containing 2 blocks

Optimisations – S-Box Rotations (SHA1)

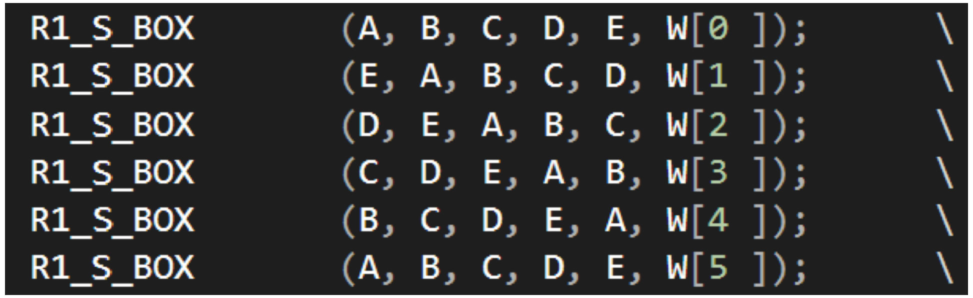
- Rotate using pre-processor macros – removes 4 assignments per S-Box (320 per SHA1 round)



```
for ( t = 0; t < 16; ++t )
{
    temp =  ROTATE_LEFT( A, 5 )          +
            OCL_BIT_SELECT( B, C, D )    +
            E                            +
            W[ t ]                       +
            K0;
    E = D;
    D = C;
    C = ROTATE_LEFT( B, 30 );
    B = A;
    A = temp;
}
```



```
#define R1_S_BOX(A, B, C, D, E, W)                  \
{                                                   \
    E = (ROTATE_LEFT(A,5)                           \
        + OCL_BIT_SELECT(B,C,D) + E + W + K0);      \
    B = ROTATE_LEFT(B,30);                          \
}
```



```
R1_S_BOX        (A, B, C, D, E, W[0 ]);      \
R1_S_BOX        (E, A, B, C, D, W[1 ]);      \
R1_S_BOX        (D, E, A, B, C, W[2 ]);      \
R1_S_BOX        (C, D, E, A, B, W[3 ]);      \
R1_S_BOX        (B, C, D, E, A, W[4 ]);      \
R1_S_BOX        (A, B, C, D, E, W[5 ]);      \
```

Cryptanalytic Optimisation Summary

- Merkle-Damgård and Key-Stretching optimisations remove ~75% of all necessary SHA1 round stages.
- Remaining SHA1 round stages benefit from the following instruction count reductions:

Optimisation	ADD	XOR	[ ]	=	CMP
Zero-Based	11				
S-Box Redundant XOR		27	27		
S-Box Rotations				320	
Cyclic Storage	▨	**64x int32 memory per kernel**			▨
HMAC Redundant Checks					2
PBKDF2 1000 loops:	11,000	27,000	27,000	640,000	2,000

GPGPU Programming Overview

- OpenCL solution, supports NVIDIA & ATI GPUs, AMD & Intel CPUs and Altera FPGAs
- GPUs have much slower clock speeds than CPUs
- Many more processing elements (stream processors / SIMD-Vector Units), 2-3k on top-end cards
- Massive memory bandwidth (ATI R9 290X – 352 GB/s)
- Manual data buffering / bus transfers
- OpenCL Kernels run on GPU, analogous to a shader program (HLSL)

GPGPU Programming Overview

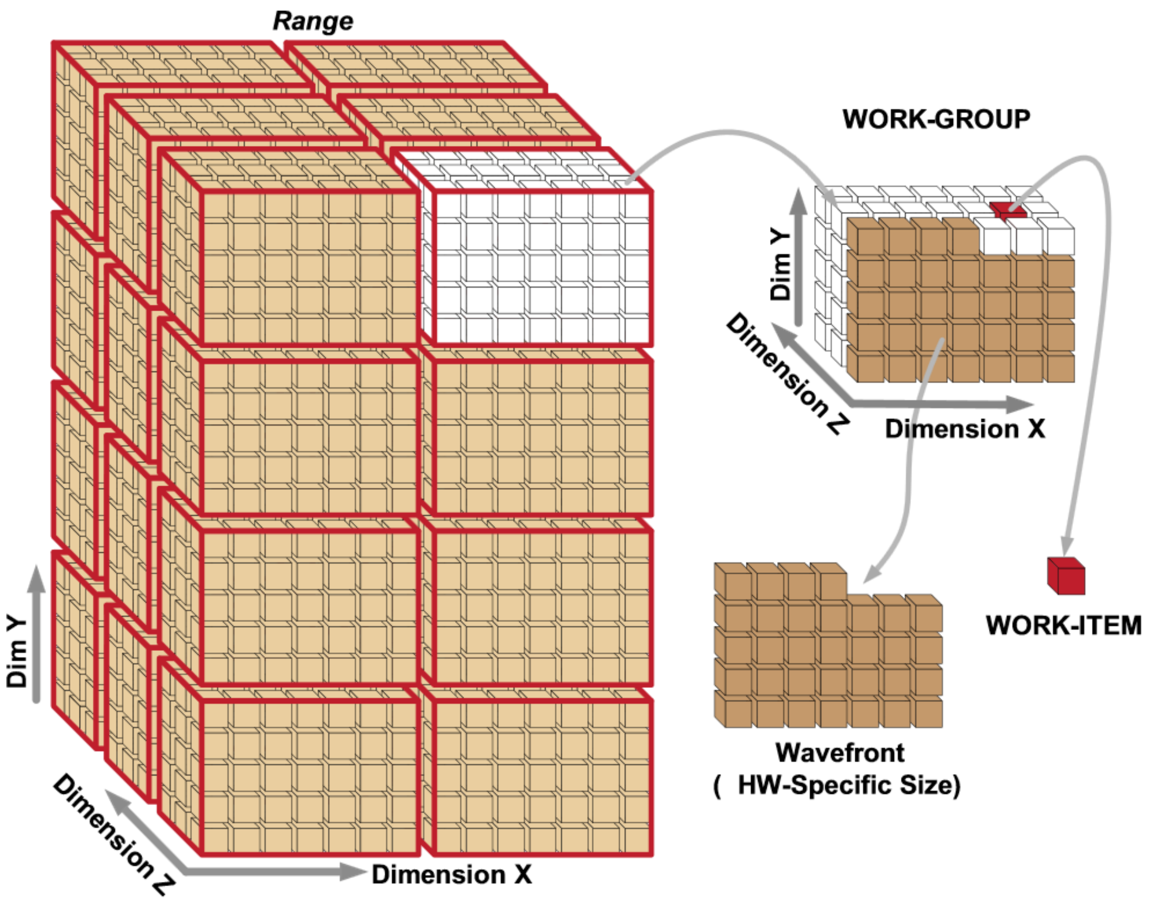


Image from AMD opencl programming guide

- The execution of a single kernel is termed a Work-Item
- Work-Items are grouped into Wavefronts (termed Warp by NVIDIA)
- Work-Group can consist of upto 4 Wavefronts
- Device Compute Units can handle multiple in-flight Work-Groups at a time.

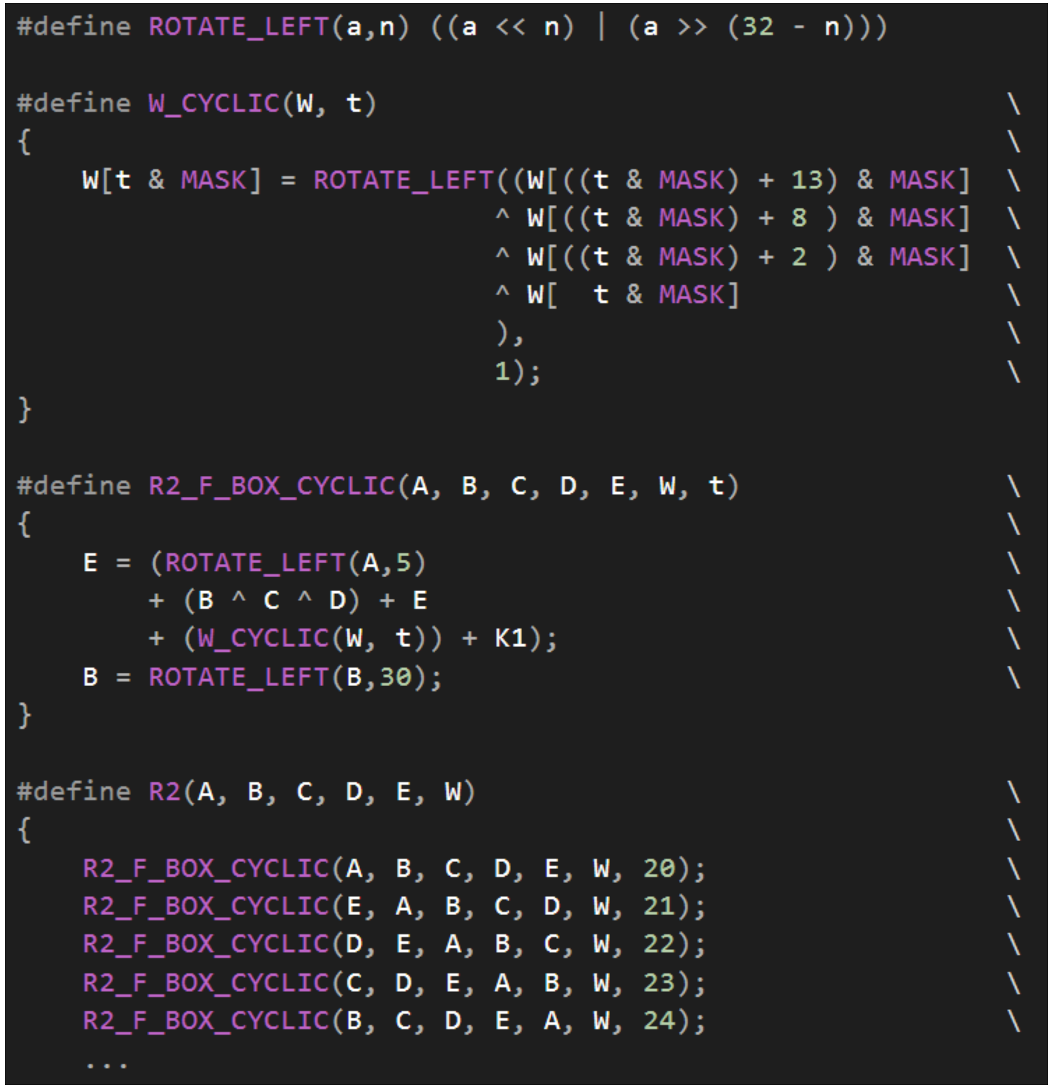
http://developer.amd.com/wordpress/media/2013/07/AMD Accelerated Parallel Processing OpenCL Programming Guide-rev-2.7.pdf

PBKDF2 Optimisations – OpenCL Kernel

- Manual unrolling / inlining
- Bus data transfers – GPU collision detection
- Occupancy / latency hiding
- Memory access coalescence
- Instruction Packing
- Work group sizes

Optimisations – Manual Unrolling / Inlining

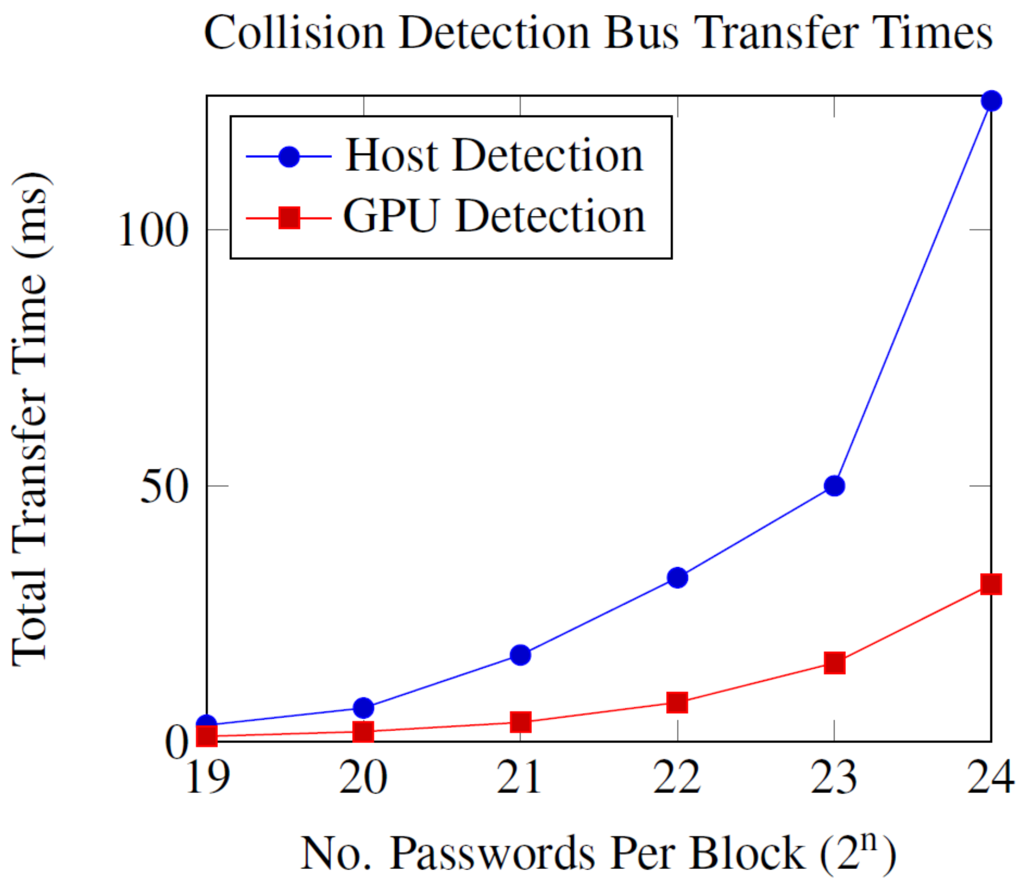
- AMD OpenCL automatic loop unrolling is not optimal
- Forces developers to work around compiler bugs
- Manual unrolling of all core loops and inlining the majority of function calls results in excess of a 70% performance gain



```
#define ROTATE_LEFT(a,n) ((a << n) | (a >> (32 - n)))

#define W_CYCLIC(W, t)                                      \
{                                                           \
    W[t & MASK] = ROTATE_LEFT((W[((t & MASK) + 13) & MASK]  \
                             ^ W[((t & MASK) + 8 ) & MASK]  \
                             ^ W[((t & MASK) + 2 ) & MASK]  \
                             ^ W[  t & MASK]                \
                             ),                             \
                             1);                            \
}

#define R2_F_BOX_CYCLIC(A, B, C, D, E, W, t)                \
{                                                           \
    E = (ROTATE_LEFT(A,5)                                   \
        + (B ^ C ^ D) + E                                   \
        + (W_CYCLIC(W, t)) + K1);                           \
    B = ROTATE_LEFT(B,30);                                  \
}

#define R2(A, B, C, D, E, W)                                \
{                                                           \
    R2_F_BOX_CYCLIC(A, B, C, D, E, W, 20);                  \
    R2_F_BOX_CYCLIC(E, A, B, C, D, W, 21);                  \
    R2_F_BOX_CYCLIC(D, E, A, B, C, W, 22);                  \
    R2_F_BOX_CYCLIC(C, D, E, A, B, W, 23);                  \
    R2_F_BOX_CYCLIC(B, C, D, E, A, W, 24);                  \
    ...
```

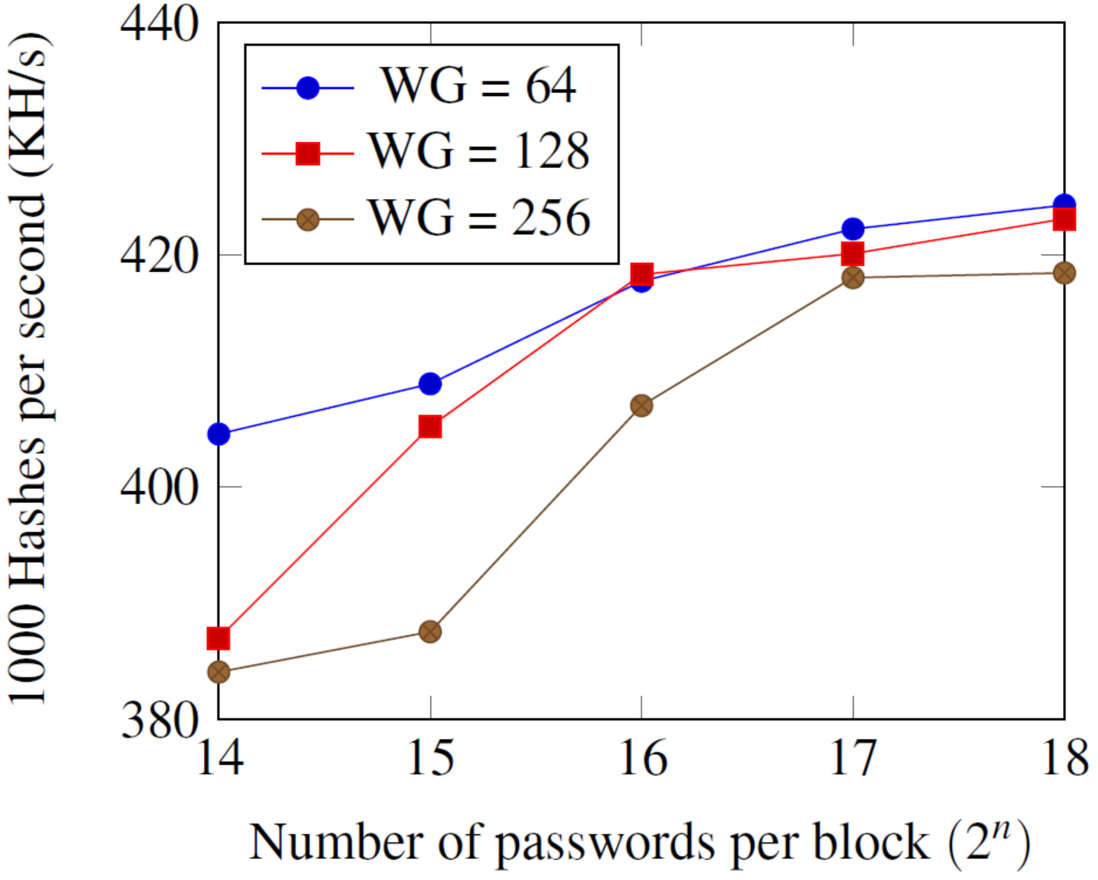
Optimisations – GPU Collision Detection

- Bus transfers are costly
- In SHA1 if all hash results are transferred back to host, this results in 22% of execution time spent serving memory requests
- Calculating hash collisions on the GPU is more efficient – we only transfer a single boolean per password block
- If a crack is found, a second buffer contains the plaintext password



Optimisations – Work Group Sizes

- If work-group size is already large enough to mask any memory access latencies, increasing WG size adds additional wavefront context switching overhead
- Optimal results were always obtained with a single WG for PBKDF2



Kernel Optimisation Summary

Optimisation	Approximate Speed Increase
Manual Unrolling / Inlining	70%
Instruction Packing	12%
Workgroup Sizes	1.37 – 5.07% (block size dependant)
Bus data transfers	0.09% (31.03% less bus memory traffic)
Occupancy / Latency Hiding	100%

Results

GPU	SHA1	HMAC-SHA1	PBKDF2-HMAC-SHA1
ATI HD6870, 1GB	794.60 MH/s	395.21 MH/s	424.78 KH/s
ATI R9 290X, 4GB	3,415.37 MH/s	1,610.62 MH/s	1611.98 KH/s

- Our PBKDF2 is 11.09% faster than oclHashCat on R9 290X
- Our HMAC is 8.5% faster than oclHashCat on R9 290X

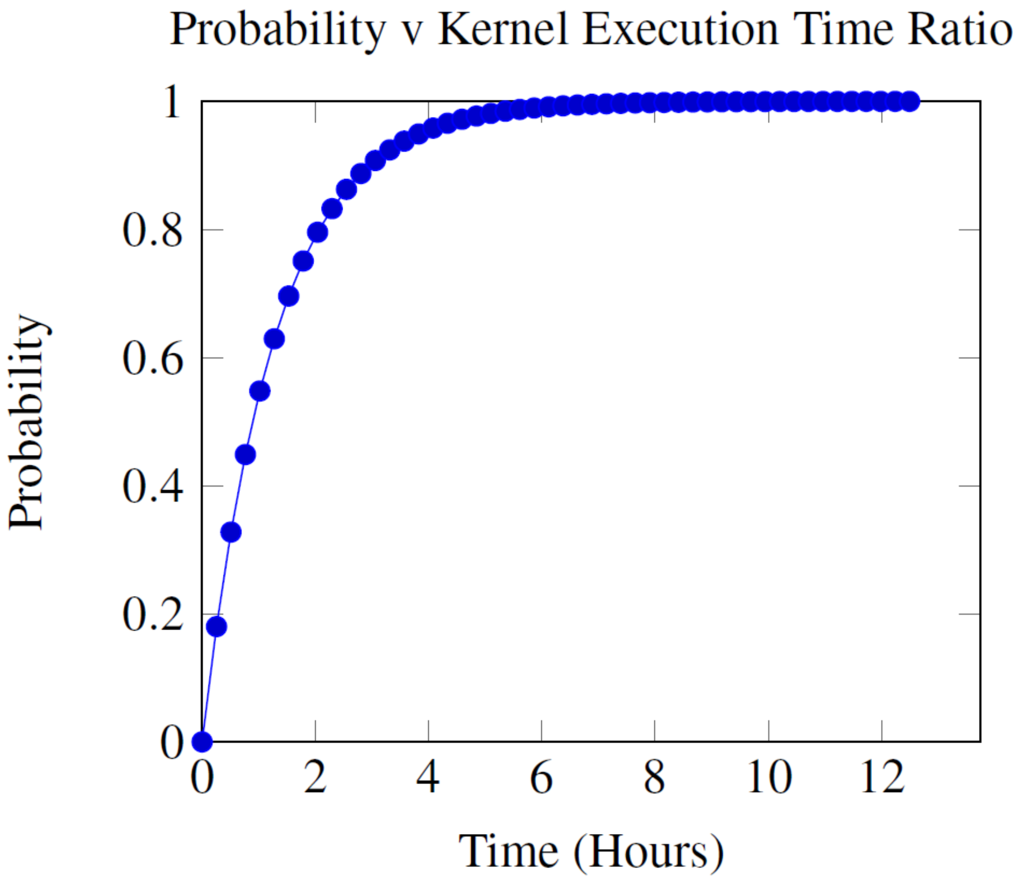
- PBKDF2 results based on 1,000 iterations and a 256 bit output key size

Cracking .NET Passwords

- ~15% of all websites worldwide run on ASP.NET
- Default password hashing uses PBKDF2-HMAC-SHA1, 1000 iterations and a 256-bit key size
- Our application provides direct support for cracking .NET hashes
- We achieve a real throughput speed of 1,608,860 passwords / sec (10.36 mins per 1 billion candidates) on an ATI R9 290X GPU
- A previous password data dump, following a security breach lead to an 18.2% crack success rate from a dictionary containing 1.494 billion words

Cracking .NET Passwords

- High probability of cracking a password after trying 10 or 11 against our dictionary
- This would take us 2.58 – 2.83 hours, on a single GPU



Application to WPA2

- Only difference in WPA2 is 4,096 iterations
- Our attack equally applies to WiFi security – 10.56 -11.59 hours to try 10 or 11 networks

Conclusions

- Cryptanalytic optimisations provide a larger contribution than hardware acceleration (measurement details see our paper)
- An optimal SHA1 ≠ optimal HMAC ≠ optimal PBKDF2
- We are now state-of-the-art for PBKDF2 and HMAC

Conclusions

- oclHashcat outperforms competitors due to their cryptanalytic optimisations, which combined with GPU acceleration made them the previous state-of-the-art
- Our PBKDF2 implementation is ~11.09% faster, thus the chance of further hidden optimisations in oclHashCats implementation is low
- Small optimisations to SHA1 = large benefits in PBKDF2

Conclusions

- Our attacks pose a real threat to actively deployed security systems, including .NET and WPA / WPA2, amongst many others
- The definition of PBKDF2 in both PKCS#5 (IETF RFC 2898) and NIST FIPS SP 800-132 contains 2 serious design flaws:
    1. Inner HMAC is incorrectly keyed; If password and salt were swapped, we'd be unable to exploit this
    2. key stretching is fundamentally broken; only ever use one block for passwords
- PKCS#5 should be updated to use H(p||s||c) as defined by Yao & Yin
- Future implementations should consider memory-hard functions

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)

Questions?

Andrew Ruddick – andrew.ruddick@hotmail.co.uk

Jeff Yan – jeff.yan@lancaster.ac.uk

Source Code: https://github.com/OpenCL-Andrew/.NETCracker/