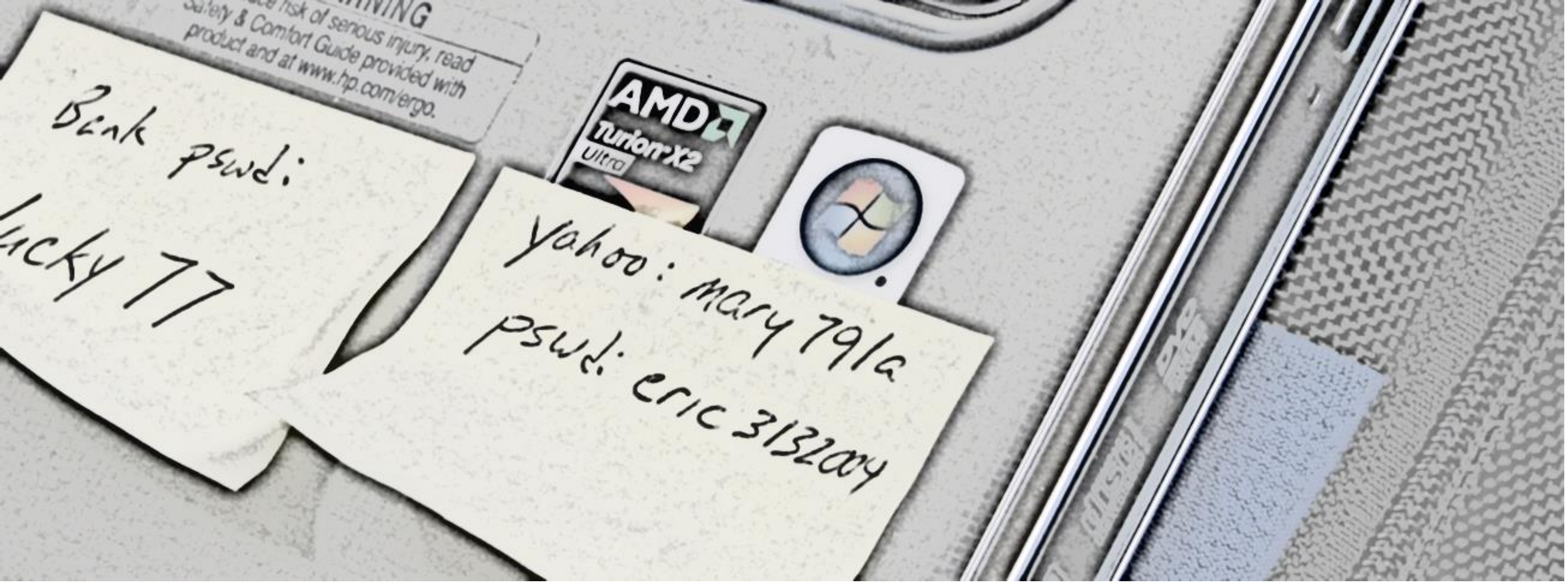


RUHR-UNIVERSITÄT BOCHUM

Methods of User Authentication: Programming Tutorial 1

Maximilian Golla

Do We Care?

Schneier on Security

Blog Newsletter Books Essays News Speaking Crypto About Me

← Information in Your Boarding Pass's Bar Code

I'm a Guest on "Adam Ruins Everything" →

SHA-1 Freestart Collision

There's a new cryptanalysis result against the hash function SHA-1:

> **Abstract**: We present in this article a freestart collision example for SHA-1, i.e., a collision for its internal compression function. This is the first practical break of the full SHA-1, reaching all 80 out of 80 steps, while only 10 days of computation on a 64 GPU cluster were necessary to perform the attack. This work builds on a continuous series of cryptanalytic advancements on SHA-1 since the theoretical collision attack breakthrough in 2005. In particular, we extend the recent freestart collision work on reduced-round SHA-1 from CRYPTO 2015 that leverages the computational power of graphic cards and adapt it to allow the use of boomerang speed-up techniques. We also leverage the cryptanalytic techniques by Stevens from EUROCRYPT 2013 to obtain optimal attack conditions, which required further refinements for this work. Freestart collisions, like the one presented here, do not directly imply a collision for SHA-1.
>
> However, this work is an important milestone towards an actual SHA-1 collision and it further shows how graphics cards can be used very efficiently for these kind of attacks. Based on the state-of-the-art collision attack on SHA-1 by Stevens from EUROCRYPT 2013, we are able to present new projections on the computational/financial cost required by a SHA-1 collision computation. These projections are significantly lower than previously

Search

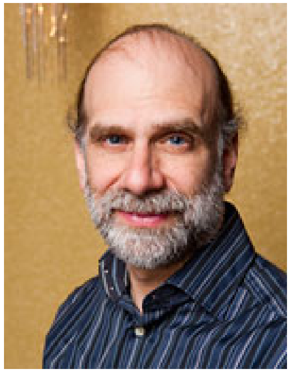
Powered by DuckDuckGo

Go

blog essays whole site

Subscribe

About Bruce Schneier



Recap: Cryptographic Hash Functions

German	English	Given	Should be Hard
Einwegfunktion	Preimage resistance	hash(X)	X
Schwache Kollisionsresistenz	2^{nd}-Preimage resistance	Y	Y' with hash(Y) == hash(Y')
Starke Kollisionsresistenz	Collision resistance	-	Z,Z' with hash(Z) == hash(Z')

Recap: The Problem

- Hash functions are “**fast**” and **deterministic**

```
$> echo "mySecurePassword" -n | sha1sum
07b5e42f1eebdafc0591f67579a9db194e2c8f97

$> echo "mySecurePassword1" -n | sha1sum
53aeab619c8fefd80c01b81451b9c1d91ce09f7f

$> echo "mySecurePassword" -n | sha1sum
07b5e42f1eebdafc0591f67579a9db194e2c8f97
```

Recap: Storing Passwords

- We do not store plaintext passwords but,
  - **salted** (e.g., 128-bit)
  - **iterated** (e.g., $2^{12} = 4096$ rounds)
  - **hashed** (e.g., **bcrypt** / scrypt / PBKDF2 / Argon2i)

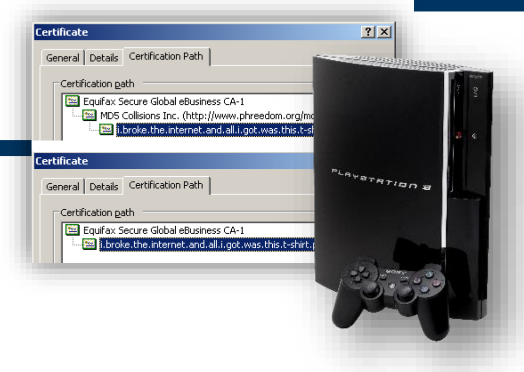
  passwords in the database.

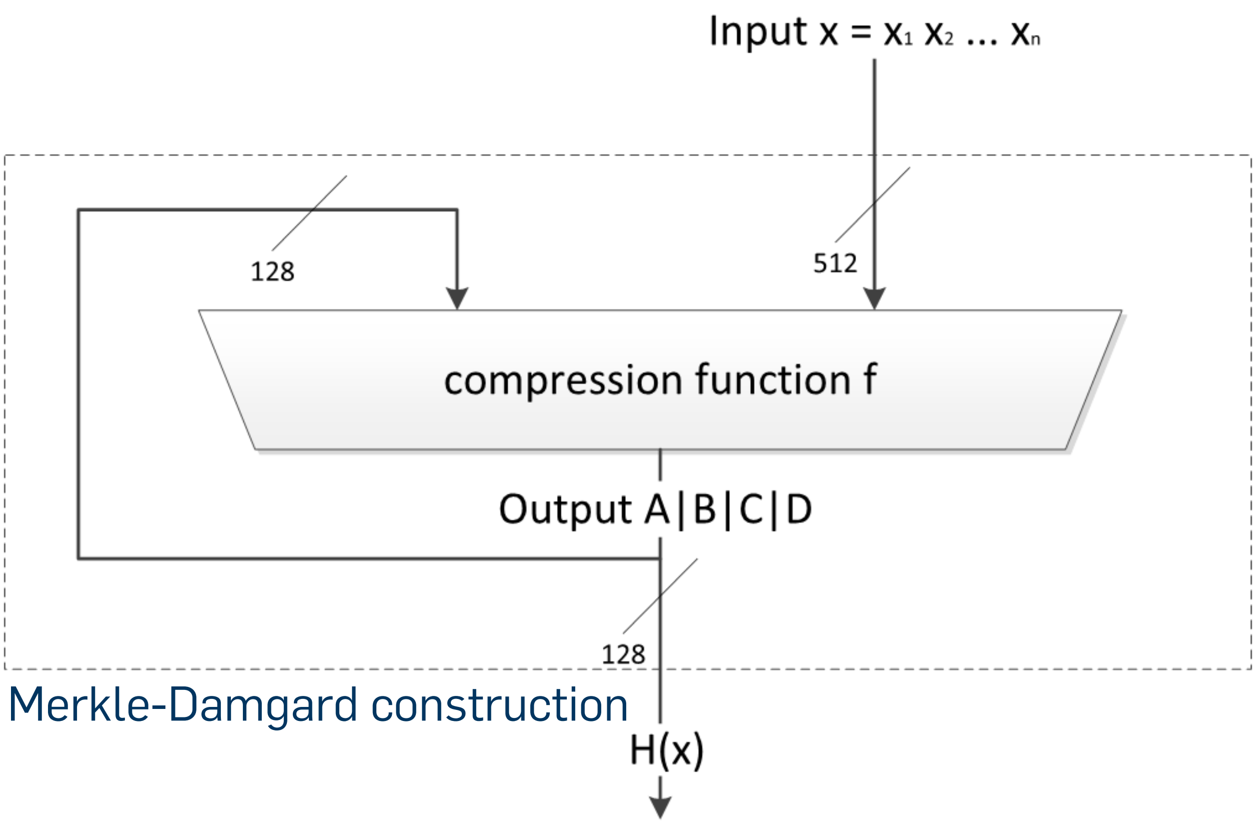
- Output string containing everything we need to know:

$2a$12$8vxYfAWCXe0Hm4gNX8nzwuqWNuk0kcMJ1a9G2tD71ipotEZ9f80Vu

Example MD5

- Ronald L. Rivest 1992, successor of MD4
- Collision resistance 2^{64} (Birthday Attack)
- Collision in 2004 (Wang et al.)
- Highly practical in 2008 (Stevens et al.)





Merkle-Damgard construction

In More Detail

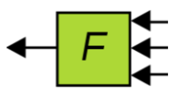
- 64 rounds (4x 16)
- Input: 512-bit, Output: 128-bit

Every single round consist of:

✓ 4x 32-bit variables 

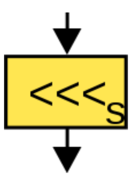
✓ M_i Message (Input string)

✓ K_i Constants (By the author)

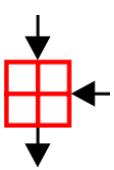
✓ 

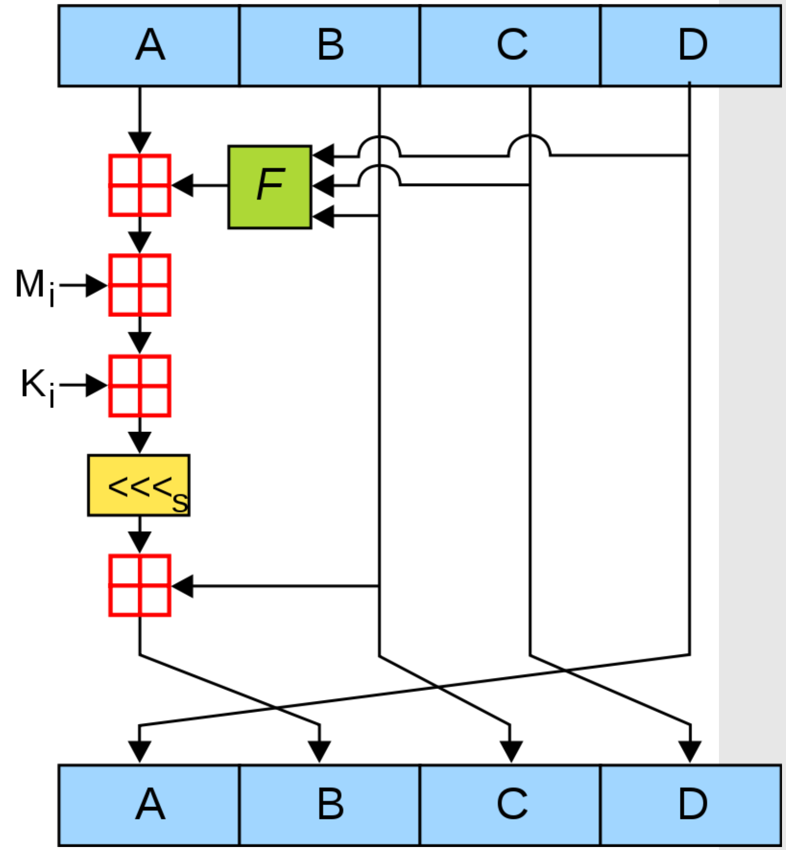
Non-linear compression function

Boolean bitwise functions (AND, OR, XOR, NOT)

✓ 

Bitwise Left Rotation (ROL)

✓  Modular addition (2^{32})

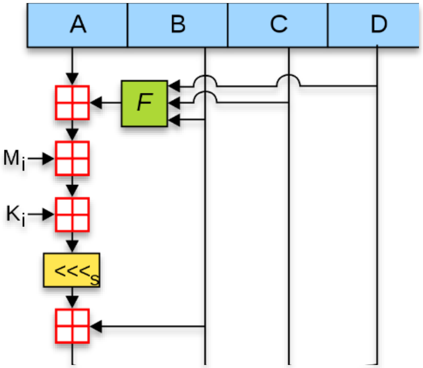


A single round of MD5

Optimizing Hash-Algorithms as an Attacker

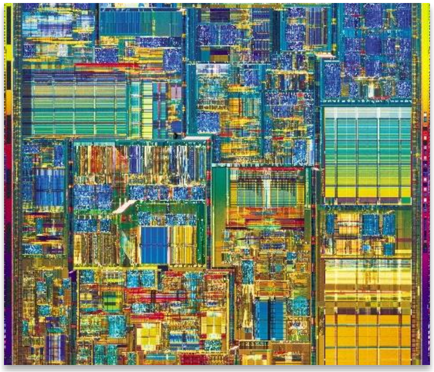
GNU Compiler

Code



Msg. Expansion Attack

Hash-based



SIMD

Hardware

How to Speed Up!

- 1. Message padding
  - Input x (Password candidate) has length b = |x|
  - Append single bit “1”
  - Fill with “0”-bits until 448 mod 512
  - Add length b as two 32-bit words (lower order first)

-> Use fixed padding

- 2. Defining constants

a0 := 0x67452301
b0 := 0xEFCDAB89
c0 := 0x98BADCFE
d0 := 0x10325476

For all i from 0 to 63:
$K[i] := \text{floor}(\text{abs}(\sin(i + 1)) \times 2^{32})$

-> Precompute

- 3. Compression functions

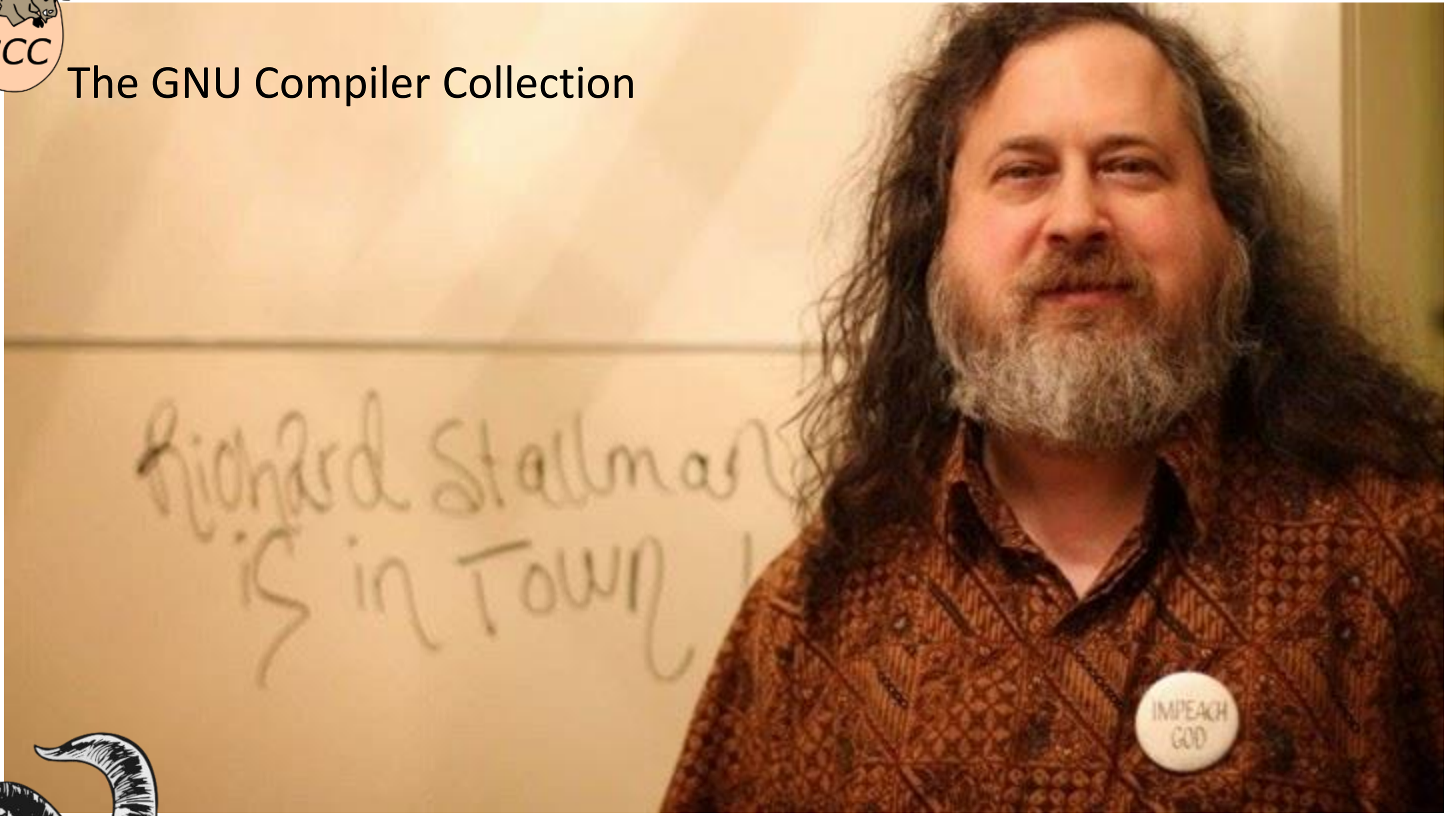
$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

-> Use optimized functions

($0 \le i \le 15$): F := D xor (B and (C xor D))
($16 \le i \le 31$): F := C xor (D and (B xor C))

Code Optimizations

The GNU Compiler Collection



Code Optimizations

- **Constants**
  Is a variable whose value cannot be changed once it is initially bound to a value. There's usually a small performance and memory advantage in using constants.

```
const __m128i K00_19 = SET1INT(0x5A827999);
```

- **Multiple Variable Declaration**
  All variables are allocated in one memory space. The count of variables can impact the performance.

```
uint32_t interm, interm2, interm3, interm4, foo, bar;
```

Code Optimizations

- **Static Inline Functions**
  Making a function an inline function suggests that calls to the function be as fast as possible. The exact situations under which an inline function actually gets in-lined vary depending on the compiler; the details of this are complicated.

```
static inline __m128i F00_19(__m128i B, __m128i C, __m128i D) {
    return XOR(D, AND(B, XOR(C, D)));
}
```

- **Macros**
  They are a way of eliminating function call overhead because they are always expanded in-line, unlike the "inline" keyword which is an often-ignored hint to the compiler, and didn't even exist (in the standard) prior to C99.

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

Code Optimizations

- **Inline Assembly**
  With inline assembly we are able to further accelerate the execution of applications by handcrafting the assembler codes for the most performance-sensitive parts.

```
__asm__("ROR %1,30\n\t" : "=a" (var) : "a" (var));
```

- **FIPS PUB 180-1 Alternative F-Functions**
  Equivalent expressions that reduce the necessary amount of instructions per invocation.

```
(0  ≤ i ≤ 19): // Original
    f = (b and c) or ((not b) and d)
(0  ≤ i ≤ 19): // Alternative
    f = d xor (b and (c xor d))
```

Code Optimizations

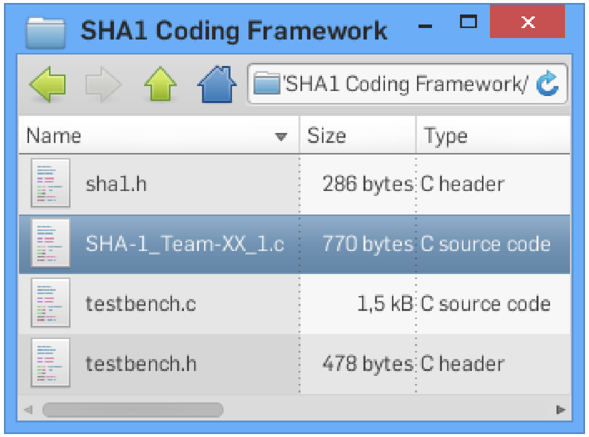
- Loop unrolling

```c
/* Normal */
int i = 0;
for (; i < 16; i++) {
    x[i] = buffer[i * 4 + 3] << 24 | buffer[i * 4 + 2] << 16 | buffer[i * 4 + 1] << 8 | buffer[i * 4 + 0];
}
/* Unrolled */
x[ 0] = buffer[ 0 * 4 + 3] << 24 | buffer[ 0 * 4 + 2] << 16 | buffer[ 0 * 4 + 1] << 8 | buffer[ 0 * 4 + 0];
x[ 1] = buffer[ 1 * 4 + 3] << 24 | buffer[ 1 * 4 + 2] << 16 | buffer[ 1 * 4 + 1] << 8 | buffer[ 1 * 4 + 0];
x[ 2] = buffer[ 2 * 4 + 3] << 24 | buffer[ 2 * 4 + 2] << 16 | buffer[ 2 * 4 + 1] << 8 | buffer[ 2 * 4 + 0];
x[ 3] = buffer[ 3 * 4 + 3] << 24 | buffer[ 3 * 4 + 2] << 16 | buffer[ 3 * 4 + 1] << 8 | buffer[ 3 * 4 + 0];
x[ 4] = buffer[ 4 * 4 + 3] << 24 | buffer[ 4 * 4 + 2] << 16 | buffer[ 4 * 4 + 1] << 8 | buffer[ 4 * 4 + 0];
x[ 5] = buffer[ 5 * 4 + 3] << 24 | buffer[ 5 * 4 + 2] << 16 | buffer[ 5 * 4 + 1] << 8 | buffer[ 5 * 4 + 0];
x[ 6] = buffer[ 6 * 4 + 3] << 24 | buffer[ 6 * 4 + 2] << 16 | buffer[ 6 * 4 + 1] << 8 | buffer[ 6 * 4 + 0];
x[ 7] = buffer[ 7 * 4 + 3] << 24 | buffer[ 7 * 4 + 2] << 16 | buffer[ 7 * 4 + 1] << 8 | buffer[ 7 * 4 + 0];
x[ 8] = buffer[ 8 * 4 + 3] << 24 | buffer[ 8 * 4 + 2] << 16 | buffer[ 8 * 4 + 1] << 8 | buffer[ 8 * 4 + 0];
x[ 9] = buffer[ 9 * 4 + 3] << 24 | buffer[ 9 * 4 + 2] << 16 | buffer[ 9 * 4 + 1] << 8 | buffer[ 9 * 4 + 0];
x[10] = buffer[10 * 4 + 3] << 24 | buffer[10 * 4 + 2] << 16 | buffer[10 * 4 + 1] << 8 | buffer[10 * 4 + 0];
x[11] = buffer[11 * 4 + 3] << 24 | buffer[11 * 4 + 2] << 16 | buffer[11 * 4 + 1] << 8 | buffer[11 * 4 + 0];
x[12] = buffer[12 * 4 + 3] << 24 | buffer[12 * 4 + 2] << 16 | buffer[12 * 4 + 1] << 8 | buffer[12 * 4 + 0];
x[13] = buffer[13 * 4 + 3] << 24 | buffer[13 * 4 + 2] << 16 | buffer[13 * 4 + 1] << 8 | buffer[13 * 4 + 0];
x[14] = buffer[14 * 4 + 3] << 24 | buffer[14 * 4 + 2] << 16 | buffer[14 * 4 + 1] << 8 | buffer[14 * 4 + 0];
x[15] = buffer[15 * 4 + 3] << 24 | buffer[15 * 4 + 2] << 16 | buffer[15 * 4 + 1] << 8 | buffer[15 * 4 + 0];
```

~~#pragma GCC optimize("unroll-loops")~~

Framework

- Can be found online in Moodle



sha1.h, testbench.c, testbench.h
(Do not touch, do not submit)

SHA-1_Team-01_1.c

- Teams of two students
- Rename file according to scheme
- Write your complete code in this file

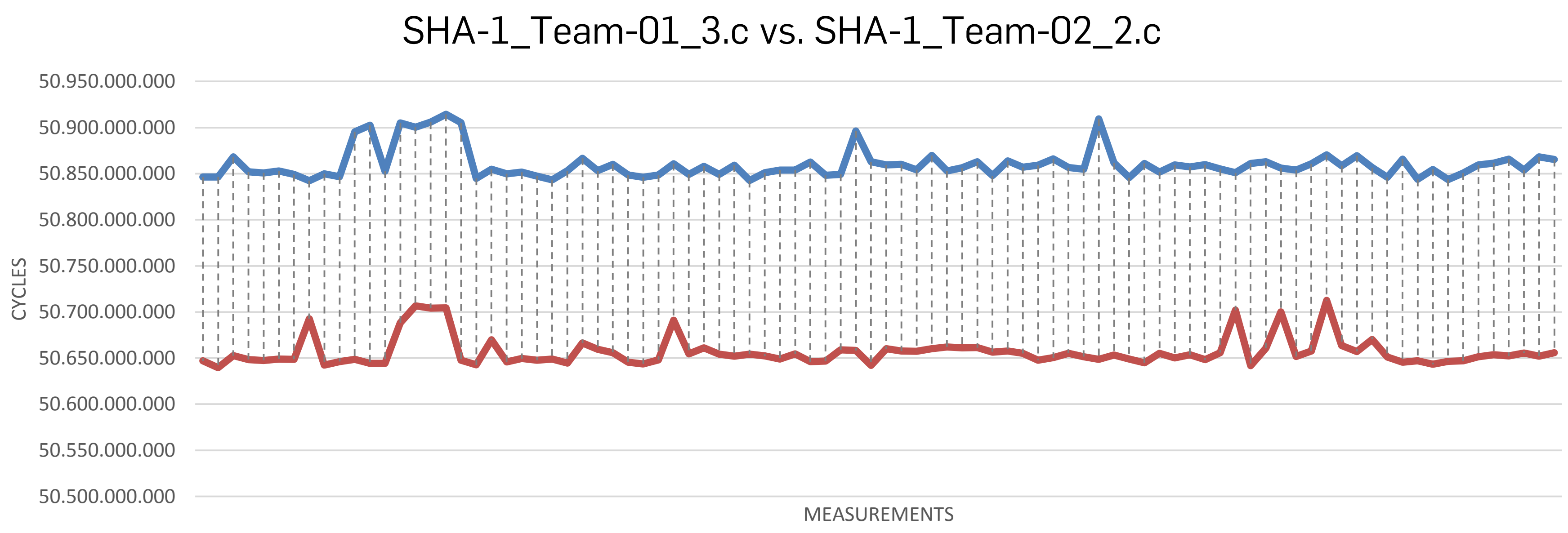
```c
#include "sha1.h"

int crackHash(struct state hash, char *result) {
    result[0] = 'a';
    result[1] = 'b';
    result[2] = 'c';
    result[3] = 'd';
    result[4] = 'e';
    result[5] = 'f';
    /* Found */
    return(EXIT_SUCCESS);
    /* Not found */
    return(EXIT_FAILURE);
}
```

```c
struct state {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    uint32_t e;
};
```

Framework

- Performance measurement via execution-time and CPU-cycles

```
clock_t start = clock();
uint64_t cnt = rdtsc();
error = crackHash(hash, result);
cnt = rdtsc() - cnt;
clock_t stop = clock();
double elapsed = (double)(stop - start) * 1000.0 / CLOCKS_PER_SEC;
```



Framework

- Example compile and execute

```
$> gcc -O2 -Wall -fomit-frame-pointer -msse2 -masm=intel
testbench.c <YOUR IMPL>.c -o crackSHA1
```

```
$> ./crackSHA1 68d8572c2662b0f06f723d7d507954fb038b8558
Hash: 0x68D8572C 2662B0F0 6F723D7D 507954FB 038B8558
Preimge: aaabbb
Cycles: 39452294775
Time: 14633.591000
```

Sneak Peek

- Next week
  - Early-Exit
  - Reverse Computation
  - Message Expansion Attack by Jens Steube (SHA-1 specific)
  - ...
  - SIMD via Intel Streaming SIMD Extensions 2 (SSE2)

Happy Coding ☺

First Round of Bonus Points - Submission Deadline:
13. November 2015 - 11:59 p.m.
Error and Warning Free!!!