Crackstation

Presented By
Nick Breese

Crackstation

"How I got my company to buy me a Playstation 3"

Presented By
Nick Breese

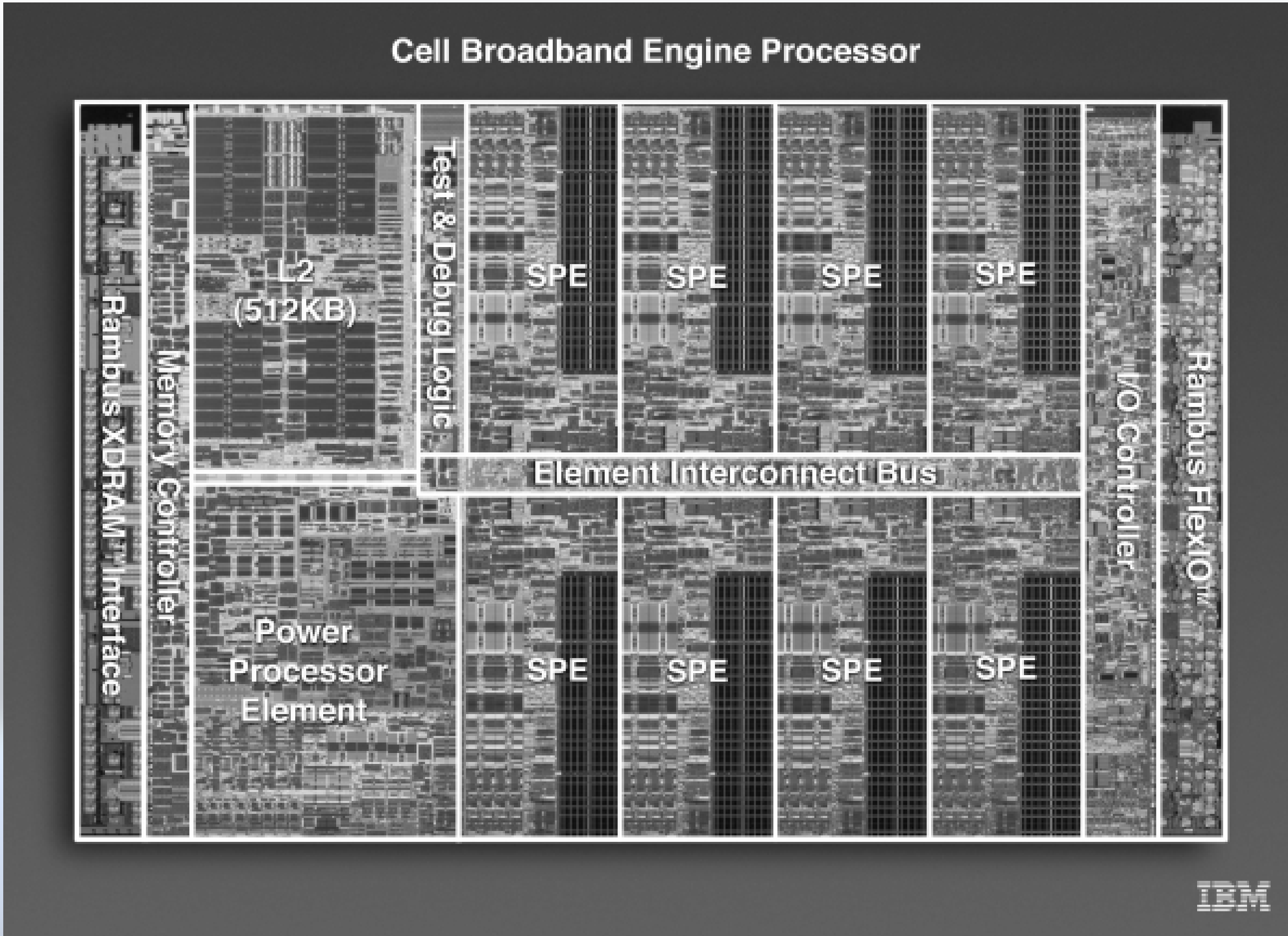
- Presentation is done in a timeline format
  - This is important as I'm not the only person to make silly assumptions
- I'm more than happy to talk about this further, however I'm very time constrained
  - Beer helps me talk more!
- All materials should be up on http://www.security-assessment.com
- Mirrored on http://insecure.io

- I wanted a Playstation 3
  - New architecture called the “Cell”
  - It would like nice on my desk
- Free toys are better than toys I have to buy myself
  - I have to convince management to buy me a PlayStation 3
- Success rates for companies buying employees game consoles is pitiful
  - Try anyway
- Used password cracking as an excuse
  - Needs lots of power
  - Sounds very “hackerish”
  - They're all for it
- The PS3 is mine!

- Features a new architecture known as the "Cell" or Cell Broadband Engine (CBE)
  - This architecture was developed by IBM, Toshiba and Sony
  - Based off IBM's Power architecture
- The PlayStation 3 is reasonably open by design
- All the developer documentation you need is publicly available
  - via IBM: http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine
- Runs custom operating systems
  - Most popular Linux distributions have some level of PS3 support
- IBM's Cell Software Development is freely available
- You have everything you need to start developing

- Yellow Dog Linux is the “official” Linux distribution supported by the Playstation 3
  - It costs money
  - I have no idea if it's any good
- IBM standardise on Fedora Core for their Cell SDK releases
  - Strongly recommend using it for development as it “just works”
  - Cell SDK 3.0 is paired for Fedora Core 7
- Installation is a little awkward. Need the following:
  - Linux distribution disc
  - Linux Add-on CD
  - Boot descriptor file (OtherOS - can be thrown on a USB key)

- At the core of the Cell is the “PPU”
  - Effectively a slightly-tweaked PowerPC core
  - PowerPC compatibility in Linux distributions make things easy
  - Be warned: using the PPU alone is relatively slow when compared to new x86 CPUs
- PPU is connected to 8 “SPUs”
  - People commonly call these the “Cell Processors”
  - These are the workhorses
  - 1 SPU is reserved for redundancy
  - 1 SPU is used for a hypervisor when using a custom OS
- In total, we have 1 PowerPC PPU and 6 SPUs at our disposal


Cell Broadband Engine Processor
Rambus XDRAM Interface
Memory Controller
L2 (512KB)
Test & Debug Logic
SPE
SPE
SPE
SPE
Element Interconnect Bus
Power Processor Element
SPE
SPE
SPE
SPE
I/O Controller
Rambus FlexIO™
IBM

- I didn't understand why the Cell is so fast. Just that it was.
  - Real-time raytracing
  - Folding@Home statistics
  - IBM “Roadrunner” supercomputer
- Don't I have everything I need now?
  - Processors are processors right?
  - I have Linux.
  - I have a custom GCC implementation for the Cell.
- Lets do it!

- Current plan:
  - Compile an MD5 implementation for the SPU
  - Use a simple wrapper to use the SPU program
  - Compare speed to other implementations
- Using L Peter Deutsch's MD5 implementation
  - It's used everywhere
  - Quite sane. Pre-computed T values used
- 10,000,000 iterations of calculating “password”
  - 8 character value

- Your custom OS runs under a hypervisor
- The GPU ("RSX") is out of bounds
  - Framebuffer is effectively all that you have available
  - Can't use the RSX to assist in cracking =(
- Aside from the loss of the RSX and one of your SPUs, you're not hindered in any way

- The SPUs run programs available within their local storage.
  - Remember: 256KB RAM only!
- It's the PPU's job to upload code to each individual SPU and trigger execution
- The Cell SDK provides two variations of GCC
  - ppu-gcc
  - spu-gcc
- We compile SPU programs with spu-gcc to create our SPU code
- We then compile a wrapper program with ppu-gcc to create our PPU code
  - It's primary purpose is to upload code to one or more SPUs and trigger execution

- The complexity is mind-boggling..

```
/* ./spu/spu-hello.c */
#include <stdio.h>
#include <spu_intrinsics.h>

int main(unsigned long long id)
{
   printf("Hello hackers from SPU ID: 0x%llx\n", id);
   return 0;
}
```

- A little more interesting..

```
/* ./ppu-hello.c */
#include <libspe2.h>
#include <stdlib.h>

extern spe_program_handle_t spu_hello;

int main()
{
  pthread_t thread[1];
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;

...
```

SPE headers

Defines our SPU program name to call

Defining variables used for our SPU context to push out

- Continued...

Tells the SPU to spawn the configured context locally

Tells the SPU to copy the program

```
ctx = spe_context_create (0, NULL);
spe_program_load (ctx,&spu_hello);
```

Create a new thread to access later

```
pthread_create (
        &thread,
        NULL,
        spe_context_run(ctx,&entry, 0, NULL, NULL, NULL), &ctx);
```

Tells the SPU to copy the program

Join the thread

```
  pthread_join (thread, NULL);
  printf("\nThe program has successfully executed.\n");
}
```

Thread finished. All done.

- Compilation

```
cd spu
/usr/bin/spu-gcc -O3 -o spu_hello.o -c spu_hello.c
/usr/bin/spu-gcc -o spu_hello  spu_hello.o
/usr/bin/ppu-embedspu -m32 spu_hello spu_hello spu_hello-embed.o
/usr/bin/ppu-ar -qcs lib_spu_hello.a spu_hello-embed.o
cd ..
/usr/bin/ppu32-gcc -mabi=altivec -maltivec -O3 -c ppu_hello.c
/usr/bin/ppu32-gcc -lspe2 -lpthread -o hello  ppu_hello.o \
   spu/lib_spu_hello.a
```

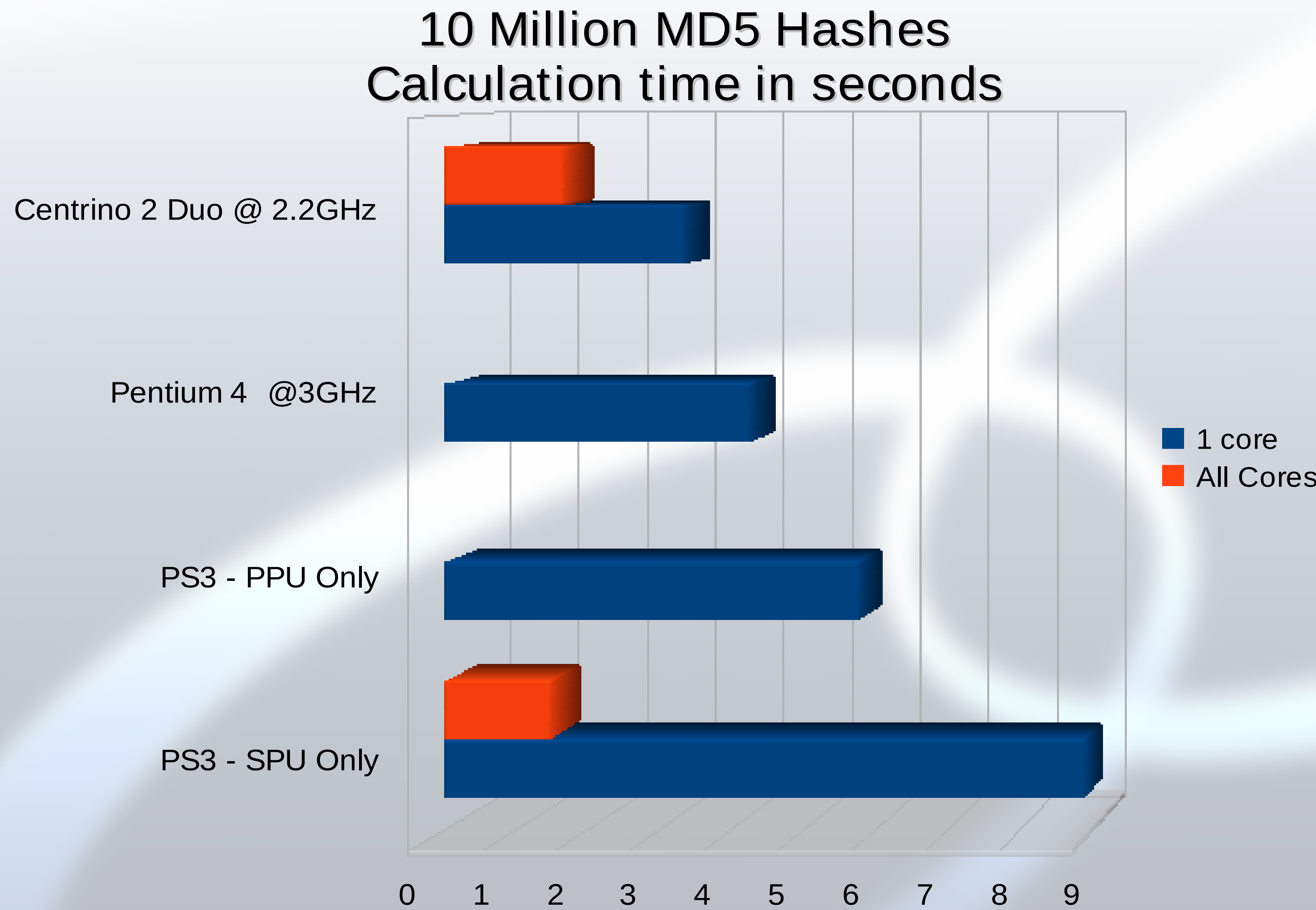
- Run-time

```
[tmasky@crackstation clean]$ ./hello
Hello hackers from SPU ID: 0x10019008

The program has successfully executed.
```

From the SPU

From the PPU


10 Million MD5 Hashes
Calculation time in seconds
Centrino 2 Duo @ 2.2GHz
Pentium 4 @3GHz
PS3 - PPU Only
PS3 - SPU Only
0
1
2
3
4
5
6
7
8
9
1 core
All Cores

- Using standard scalar code, the entire PS3 performs about as fast as one of the latest Intel Centrino processors
  - By itself, a single SPU core is the **slowest** tested architecture
- This is obviously lacking in the magnitudes of performance I was expecting
- Well okay, this is simply lame
  - How can I optimise this a bit?
  - Or, how can I explain to management this “super computer” is as fast as my laptop for crypto?

- I decided to do some reading about the features of the SPU
- It's funny what you can learn from reading things
- My initial assumption regarding the SPU processor was completely wrong
  - It's basically a vector processor, rather than your typical CPU
  - Makes “SIMD” operations run really quickly
  - Scalar support is there for convenience only
- I'm still lost...

- This is old and new tech
  - Old: The technology (vector computation) has been around for a while
  - New: It hasn't been generally exploited in public cryptography implementations
- Jump on in, it's easy
- For those not wanting to buy a PS3, the technique I will be talking about is available on other platforms including x86
  - x86's implementation is known as “SSE”
  - SSE is the evolution of MMX
  - Finally, you get to understand what MMX/SSE is
- You learn a hell of a lot about processor architecture, distributed computing, cryptography, vector and parallel processing

- This is the name for the stuff we're all familiar with
  - Like adding 1 + 1
  - i.e.:

```
int i;
i=1+2;
```

- Simple and easy
  - Single data groups (1 and 2)
  - Single operation (addition)
  - An operation is performed on a single data item at a time

- The commonly used term is “SIMD”
  - Single Instruction, Multiple Data
  - Conducting one operation against a range of values
- Example:
  - 1+4, 2+4, 3+4, 4+4
  - 4 data objects (1,2,3,4)
  - Single operation (+4)

- Single instruct, single data (scalar)
  1 + 4 = 4
  1 data + instruction
- Single instruction, multiple data (vector)
  {1, 2, 3, 4} + 4 = {5, 6, 7, 8}
  1, 2, 3, 4 data set
  + single instruction
- Single instruction, multiple data (vector)
  {1, 2, 3, 4} + {5, 6, 7, 8} = {6, 8, 10, 12}
  1, 2, 3, 4 data set
  + single instruction
  5, 6, 7, 8 second data set
- SIMD allows you to perform instructions on data sets, rather than a single piece of data

- The SPUs conduct these vector instructions natively
- This is where I got quite interested
  - I now understand why the initial basic port was so slow
  - I also have this SIMD concept to play with
- Could I write an algorithm using only vector instructions?
- Could I conduct multiple operations simultaneously to further increase speed?
- I can't hack/optimise an existing implementation, I have to write mine from scratch

- I looked at a few crypto algorithms
  - Must not be too complex
  - Must be commonly used
- MD5 was the winner
  - Met the above criteria
  - Everyone (incorrectly) uses MD5 hashes for security

- Explicitly uses little-endian 32bit unsigned integers for everything
  - x86 = little-endian (hex F2 == 0x000000F2)
  - PowerPC/SPU = big-endian (hex F2 == 0xF2000000)
  - Unsigned means only positive values
- The algorithm focuses on pre-defined starting values of 4 32-bit words
  - “a”, “b”, “c”, “d”
- Has 64 constant “T” values are fed into calculation to mix things up
  - Pre-compute the value and hard-code them. (Simple optimisation)
- Operates on blocks of 512-bits of data
  - This includes some padding
- Splits the block into 16 32-bit chunks

- *a,b,c,d* tumble through 4 rounds
- Each round is conducted 16 times
- During each step:
  - A T value is thrown in
  - A 32bit chunk of our data is also thrown in
  - Addition and bitwise operations conducted (XOR, NOT, AND, etc.)
- At the end, the current values of *a,b,c,d* are added to the initial values of the same variables
- Each 32-bit value of *a,b,c,d* is compounded to result in a 128-bit hash.

- Everything uses unsigned 32bit integers
- There are 68 predefined values that we reference
  - 64 constant T values
  - 4 constant initial values (a,b,c,d)
- Very simple maths is done
- Bitwise operations are done
- We can't split up the operations as each operation is interdependent.
  - i.e. an avalanche occurs

- The SPU has 3 methods for immediate data storage
  - The system RAM (relatively slow to access)
  - The local SPU RAM (small, fast)
  - The register space on the SPU itself (really, really fast)
- CPUs have registers too. They're temporary slots for doing immediate calculations.
  - Data is shifted from RAM into these registers, operations are performed and then the data is moved out of these registers.
- So keeping and using data within the processor's registers for as long as possible == speed
- x86 architecture has 8 – 16 general purpose 32-bit registers, 8 – 16 vector registers
- So, what about the SPU?
  - **128!**

- Each vector register on the SPU is 128-bits wide
- When you create a vector, you can use different data lengths to a total length of 128bits. Examples:
  - 16 8-bit values
  - 4 32-bit values
  - 2 64-bit values

Back to the SPU - omg

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12	byte 13	byte 14	byte 15
doubleword 0								doubleword 1							
fullword 0				fullword 1				fullword 2				fullword 3			
halfword 0		halfword 1		halfword 2		halfword 3		halfword 4		halfword 5		halfword 6		halfword 7	
char 0	char 1	char 2	char 3	char 4	char 5	char 6	char 7	char 8	char 9	char 10	char 11	char 12	char 13	char 14	char 15

- SIMD vector operations are a simple concept
- Establish a vector with values
  - e.g. Vector A contains: 0x11111111,0x22222222,0x33333333,0x44444444
- Apply the operation against the vector with another value (could be another vector..)
- New vector is created with the resulting values.
- SPU supports all your basic math and (most) bitwise operators natively

- Review of our previous MD5 findings
- Everything uses unsigned 32bit integers
  - So, we use vectors containing 4 32-bit unsigned integers
- There are 68 predefined values that we reference
  - We have lots of available registers. Use vectors for everything
- Very simple maths is done
  - SPU natively supports that
- Bitwise operations are done
  - All supported by intrinsics, except one.
- We can't split up the operations as each operation is interdependent
  - Still a problem

- Why don't we run individual md5 calculations simultaneously?
- A, B, C, D are each vectors of 4 32-bit words
- At each step of md5, we merely conduct the operation against a vector of numbers rather than a single number.
- That allows us to conduct 4 simultaneous MD5 calculations per SPU core.
  - That's 24 concurrent calculations in a Playstation 3
- So, that's at least 4 times the performance?

- Vector operations using the Cell is damn easy with IBM's SDK.
- The term is using “intrinsics”
  - Close-to-assembly level of operations
  - Very well documented
- Examples
- Scalar
  - `unsigned int i = 1;`
- Vector:
  - `vector unsigned integer vec_i = {1,2,3,4};`
- (Alternative) Vector:
  - `vec_uint_32 vec_i = {1,2,3,4};`

- More examples:
- Scalar:
  - `i = i + 1;`
- Vector:
  - `vec_i = spu_add(vec_i,1);`
- Scalar:
  - `i = i ^ 1;`
- Vector:
  - `vec_i = spu_xor(i,1);`

- md5.c
```
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
```

- spu_md5.c
```
vec_uint4 rotate_left (vec_uint4 * vec_x, unsigned int n) {
  int rshift = 32-n;
  return spu_or(spu_sl(*vec_x,n),spu_rlmask(*vec_x,-(rshift)));
}
```

- Negative value of 32-n is required to emulate the right shift when using spu_rlmask

- md5.c

```
#define F(x, y, z) (((x) & (y)) | (~(x) & (z)))
```

- spu_md5.c

```
vec_uint4 f_round_1(vec_uint4 * vec_x, vec_uint4 * vec_y, \
vec_uint4 * vec_z)
{
  vec_uint4 vec_f;
  vec_uint4 vec_f1;
  vec_uint4 vec_f2;
  vec_f1 = spu_and(*vec_x,*vec_y);
  vec_uint4 vec_comp_x = spu_nor (*vec_x,vec_null);
  vec_f2 = spu_and ( vec_comp_x, *vec_z );
  vec_f = spu_or ( vec_f1, vec_f2 );
  return(vec_f);
}
```

- Long, drawn-out programming style

- The "NOT" bitwise operator
  - Inverts the bit values
- No "spu_not"!
  - Scalar:
    `d = ~c`
  - Vector:
    `vec_d = spu_eqv(vec_c,vec_null);`
  - Vector:
    `vec_d = spu_nor(vec_c,vec_null);`
- vec_null = {0x00000000, 0x00000000, 0x00000000, 0x00000000};

- SPUs are like 100 meter runners
- They can go really fast in a straight line
  - Don't put hurdles in their way
- Branch conditions trip them up
  - Calculate both possibilities if possible
  - Select the answer to use based on a condition
  - Use spu_select with a comparison intrinsic
- Keep as much of your code linear

- So, how fast is my MD5 vector implementation on a Playstation 3?

- ~~1.4~~ 1.9 billion MD5 calculations a second

- Calculation time != cracking time
- This level of performance brings new problems
- How do you feed enough data in for the implementation to chew through?
- I don't believe you can
  - Need to instruct the SPU to generate it's own test values
  - Which happens to adhere to the principle of keeping the SPU as independent as possible

- Not specific to Cell.
  - It's only the strongest SIMD implementation out there.
- Other implementations:
  - MMX/SSE (x86)
  - Altivec (PowerPC)
  - GPUs
- I haven't seen any implementations of this technique anywhere else.

- SSE uses the same width for vector registers
  - 128bits wide
  - Supports 4 x 32bit unsigned integers
- Took 2 hours to port.
  - No prior SSE experience
  - Just picked up an SSE reference document
  - Search/replace for most things
- Roughly double the performance.

- Finding collisions is likely to be easier
- Salted password cracking
- Complex password cracking
- “What about my Linux?”
  - While distributions typically use an MD5-based password hashing mechanism, it's not just plain MD5
  - The biggest defense is that passwords are hashed through 1000 MD5 iterations
  - But the gap is closing
  - New hash needed, more iterations

- “What about my Windows?”
  - Windows revision up until Vista store LM hashes – a very weak DES-based implementation
  - The native alternative is NTLM hashes. This is MD4
  - Just to note: NTLM hashes uses UTF-16 for password data

- You would think I'm rather happy with the Cell
- Unfortunately due to limited availability, it complicates things
- Using this technology for defense is difficult
  - Though I think having PS3s hooked up to servers in datacentres would be pretty sweet.
- Ongoing security research is difficult due to accessibility of faster Cell processors
- In my opinion, it is incorrectly marketed leading to non-widespread availability
- Latest revision of the Cell shrunk the size of the die, but offered no improvements
  - It's starting to look like the Cell is the “PS3 processor”

http://www.security-assessment.com

nick.breese@security-assessment.com