



# Crackstation

Optimised Cryptography On The PlayStation 3

Presented By  
Nick Breese



# Crackstation

“How I got my company to buy me a Playstation 3”

Presented By  
Nick Breese

- I'm Nick Breese
  - From Wellington, New Zealand (are there any other kiwis here?)
- Have been doing this research for 9 months now
- "Security consultant" for Security-Assessment.com
  - We break things for a living
  - Due to security being a small market in a small country, we need to be "jack of all trades"
- Company spiels are boring

- Presentation is done in a timeline format
  - This details my progress throughout my research on this topic
- I'm more than happy to talk about this further, however I'm very time constrained
  - Beer helps me talk more!
- There is still a lot to look at. However, this presentation is a decent primer and explains what I have looked at so far
- 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!

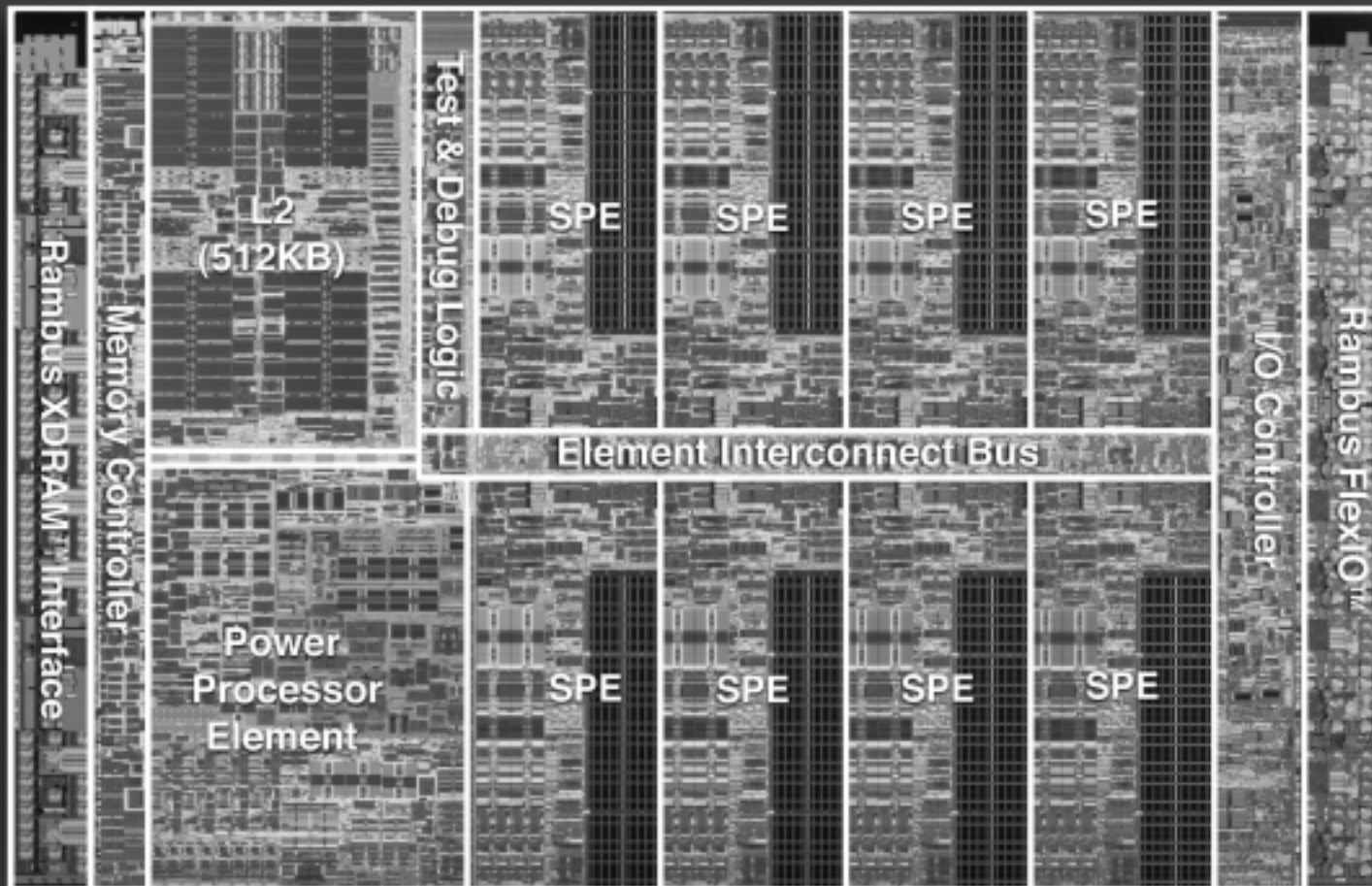
- Features a new architecture known as the “Cell” or Cell Broadband Engine (CBE)
  - This architecture was developed by IBM, Toshiba and Sony
  - Based off IBM's Power architecture
- The PlayStation 3 is reasonably open by design
- Custom code is available via running Linux on PS3
- All the developer documentation you need is publicly available
  - via IBM:  
[http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell\\_Broadband\\_Engine](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



- Your custom OS runs under a hypervisor
- The GPU (“RSX”) is out of bounds
  - I assume this is to prevent “non-PS3” games from being run on the PS3.
  - 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 SPU's, it appears you're not hindered in any way

- 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!

- A working cryptographic algorithm which utilises all available processor cores
- What Cell processor-specific technical knowledge I would need
- Look at the practicality of different cracking implementations
  - Strengths and weaknesses of this technology
- Optimise code to achieve greater performance than a workstation
- Or just generally:
  - How to do efficient cryptography on the PlayStation 3.

- 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

- The SPUs run programs available within their local storage.
  - Remember: 256KB RAM only!
- PPU uploads code to each individual SPU and triggers 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%11x\n", id);  
    return 0;  
}
```

- A little more interesting..

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

SPE headers

```
extern spe_program_handle_t spu_hello;
```

```
int main()
```

Defines our SPU program name to call

```
{  
    pthread_t thread[1];  
    spe_context_ptr_t ctx;  
    unsigned int entry = SPE_DEFAULT_ENTRY;
```

Defining variables used for our SPU context to push out

```
...
```



- Continued...

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

Tells the SPU to spawn the configured context locally

Tells the SPU to copy the program

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 execute 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
```

From the SPU



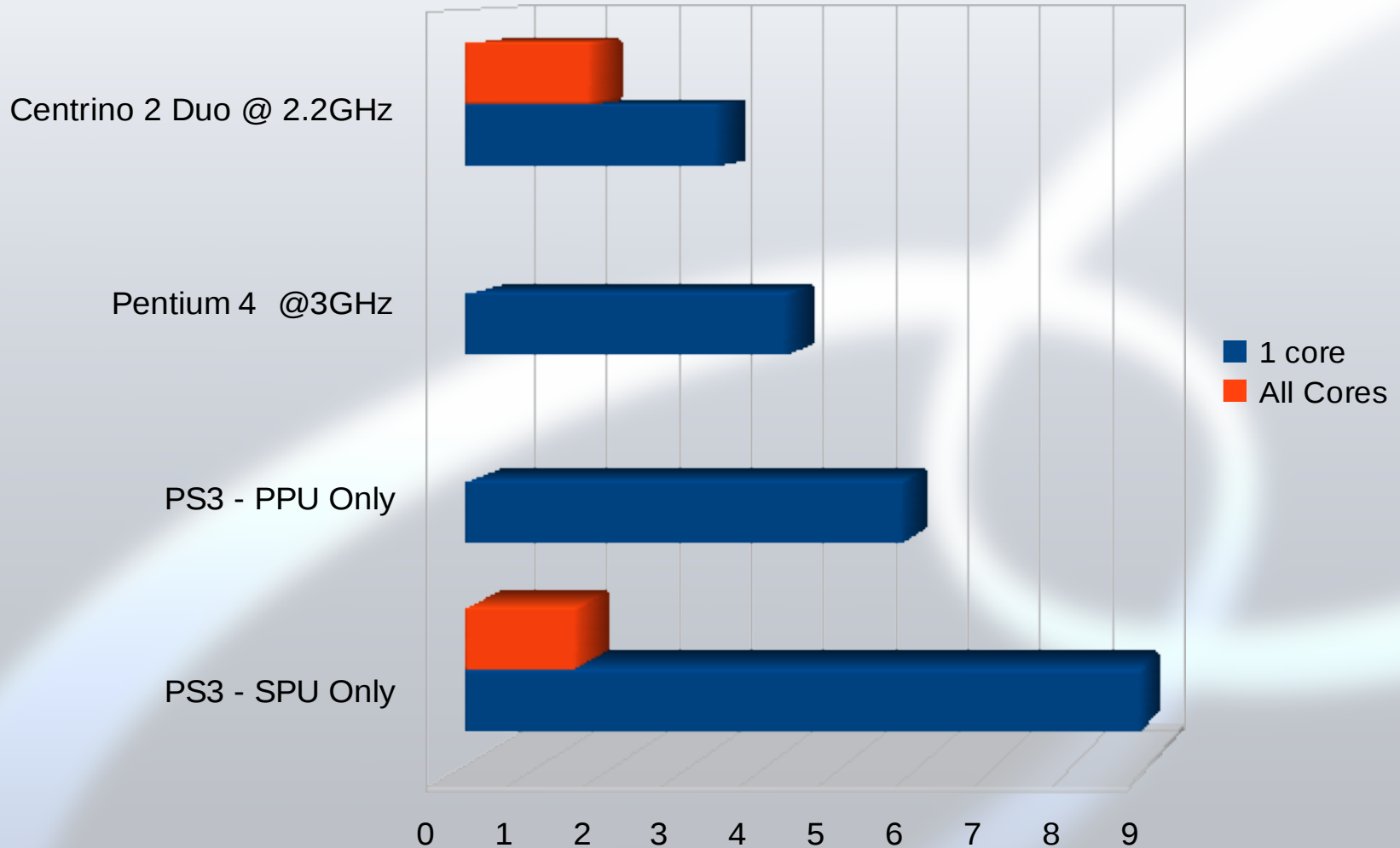
The program has successfully executed.

From the PPU



- Let's start somewhere
- Current plan:
  - Focus on the SPUs
  - Compile an MD5 implementation for the SPU
  - Use a simple wrapper to use the SPU program
  - Compare speed of the implementation on different processor architectures
- 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

## 10 Million MD5 Hashes Calculation time in seconds



- 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...

- “SIMD” vector computing 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  $1 + 2$
  - 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:
  - $1+4, 2+4, 3+4, 4+4$
  - Single instruction (+)
  - Multiple data objects  $(1,2,3,4)$  (4)

- Single instruction, single data (scalar)  
 $1 + 4 = 5$   
1 data, 4 data + instruction
- Single instruction, multiple data (vector)  
 $\{1, 2, 3, 4\} + 4 = \{5, 6, 7, 8\}$   
1, 2, 3, 4 data set  
+ single instruction  
4 data set
- 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 MD5 benchmark for the SPU 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/optimize an existing implementation, I have to write my own vector-based MD5 implementation from scratch.
  - I couldn't find any that existed.

- Explicitly uses little-endian 32bit unsigned integers for everything
  - x86 = little-endian (hex EAF240CC stored as EA F2 40 CC)
  - PPU/SPU = big-endian (hex EAF240CC stored as CC 40 F2 EA)
  - 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 general purpose 32-bit registers. 8 – 16 MMX/SSE vector registers
- So, what about the SPU?
  - **128** vector registers

- 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



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

(c) IBM

- 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.
- The entire MD5 calculation cannot be split up
  - Each part of the calculation is dependent on previous values
  - 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
- If it's a native vector processor; then it should be at least 4 times the performance of the scalar implementation on the SPU?

- #define F(x, y, z) ((x) & (y)) ...
- Standard scalar ( x & y )
  - x == 0xFFDDEECC  
y == 0x00112233
- Vector ( x<sub>0</sub> & y<sub>0</sub> ), ( x<sub>1</sub> & y<sub>1</sub> ), ( x<sub>2</sub> & y<sub>2</sub> ), ( x<sub>3</sub> & y<sub>3</sub> )
  - x<sub>0</sub> == 0xFFDDEECC  
y<sub>0</sub> == 0x00112233
  - x<sub>1</sub> == 0xEFEDEEEC  
y<sub>1</sub> == 0x10111213
  - x<sub>2</sub> == 0xDFDDDEDC  
y<sub>2</sub> == 0x20212223
  - x<sub>3</sub> == 0xDFDDDEDC  
y<sub>3</sub> == 0x30313233

- 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_uint4 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
- You can do the multiple calculations in-line, but I find it too hard to follow due to the length

- The “NOT” bitwise operator
  - Inverts the bit values
- No “spu\_not”!
  - Scalar:  
 $d = \sim c$
  - Vector:  
`vec_d = spu_eqv(vec_c, vec_null);`
  - Vector:  
`vec_d = spu_nor(vec_c, vec_null);`
- `vec_null = {0x00000000, 0x00000000, 0x00000000, 0x00000000};`

- md5.c

`((x) & (y)) | (~(x) & (z))`

AND            OR            NOT            AND

`spu_and    spu_or    spu_eqv    spu_and`

- SPU's 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_sel` with a comparison intrinsic
- Keep as much of your code linear

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

## **If you play with bleeding-edge stuff, you're likely to get cut.**

- The vector-based benchmark supposedly pulled in 1.4 to 1.9 billion calculations a second
  - This is incorrect
- About 80 million calculations per second is the current “cracking” rate
  - I will update this presentation once I have identified the cause of the significant difference in calculations times.

- The MD5 benchmark conducts repetitive calculation of a single value
  - The result is not used in any way
- The high calculation rate is most likely due to a compiler optimisation which caught me out
- I'm happy in that it is still about 20 times the performance of the initial implementation on a Core 2 Duo processor (using both cores)
- I'm not happy that I previously touted an incorrect higher number

- This level of performance brings new problems
- How do you feed enough data in for the implementation to chew through?
  - Calculation rate \* required test data ("potential password")
  - 80 million x 8 bytes = 400,000,000 bytes a second
  - If I had 200MB of available system memory, I could store enough potential passwords for 0.5 seconds
  - You're likely to saturate the SPU / system memory bus
- Simply - I don't believe you can
  - Need to instruct the SPU to generate it's own test values
  - Which makes sense, as this approach adheres to the principle of keeping the SPU as independent as possible



- Wordlists are commonly used for cracking implementations
  - Keeping each SPU independent unfortunately prohibits us from using wordlists
- We're effectively limited to simply incrementing values
  - Not necessarily a bad thing. It will catch complex passwords that wordlist-based implementations won't
  - However, the relevant printable characters are 0x20 (space) to 0x7F (~)
  - Incrementing bytes without any decent logic would generate passwords where only ~37% would be completely ASCII printable characters
- I've written a more efficient incrementer which pushes the range up
  - 0x7E -> 0xFF, 0x20->0xA1
  - Subtracts 0x81 from the range to put it back into correct range

- Branching can destroy performance
  - `spu_bisled` can execute a function if an “event” is triggered
  - Automatic branch hint to assume the function will not be called
- Be wasteful when appropriate
  - For example: incrementing character length of password. Finished testing 7 character passwords, now want to start testing 8 character passwords.
  - A: You could write complex logic to manage password length
  - B: Or you can re-run the program or execute another program that is then tailored for the character length
  - A means many more operations conducted by each iteration. B means very few, or at all.

- 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
- I've been pointed to an ASM implementation of this by another security researcher. However, I have not had the time to audit and analyse this implementation.

- 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. My implementation on the PS3 has reduced the required time to 1/20<sup>th</sup>
  - Other implementations like the NSA@Home FPGA cracker and Elcommsoft's GPU-enhanced implementation also reduce the required time

- 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
  - However, with the PS3 decreasing in price, scaling to multiple units may be effective.
- 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"

- Vector-based cryptography implementations can greatly improve calculation time
  - You are not safe if you only salt your secret
  - Linux's MD5 crypt implementation needs to be revised
  - Do we need a hashing standard that is intentionally slow to compute for passwords?
- The Cell processor could be one of the links in the chain of “next-gen” password cracking
- Keep an eye out for similar technology that could be leveraged to achieve fast cryptographic computation
  - We have FPGA, GPU and the Cell. What's next?
- Passphrases are better than passwords



<http://www.security-assessment.com>  
[nick.breese@security-assessment.com](mailto:nick.breese@security-assessment.com)