GPU Powered Malware

Daniel Reynaud

LORIA - Nancy - France

Ruxcon 2008

Motivation

- GPGPU (General Purpose programming on Graphics Processing Units) is no longer an obscure area
- Most consumer hardware is now fully programmable in C
- No need to be a specialist to tap into the computing power of GPUs
- What if malware authors start coding on GPUs ?

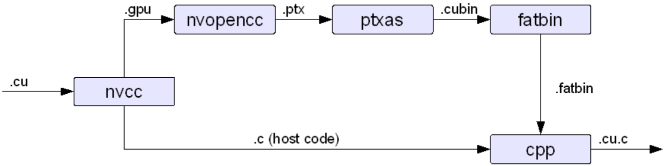
Outline

1. GPGPU Technologies
   - CUDA
   - Stream Computing
   - OpenCL
   - Larrabee
2. How could that be used in a malware ?
3. Reverse Engineering
   - Disassembling
   - Debugging
   - Emulation
4. Packing
5. Conclusion

NVIDIA's Compute Unified Device Architecture

- Requires recent NVIDIA hardware with a CUDA driver
- Easily programmable with an extension of the C language
- The device code is compiled to an assembly intermediate language, PTX and then assembled in the cubin file format (undocumented)

Here is the simplified compilation process:

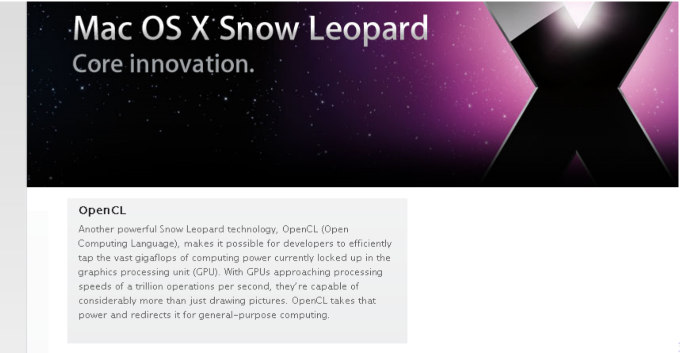


AMD's Stream Computing

- Requires recent ATI/AMD hardware with a Stream Computing driver
- Easily programmable with an (other) extension of the C language
- The device code is compiled to an (other) assembly intermediate language, AMD IL

Apple's OpenCL

- Submitted as a standard by Apple, supported by everybody except Microsoft
- Will be shipped with Mac OS X Snow Leopard
- No reference/documentation for the moment



OpenCL

Another powerful Snow Leopard technology, OpenCL (Open Computing Language), makes it possible for developers to efficiently tap the vast gigaflops of computing power currently locked up in the graphics processing unit (GPU). With GPUs approaching processing speeds of a trillion operations per second, they're capable of considerably more than just drawing pictures. OpenCL takes that power and redirects it for general-purpose computing.

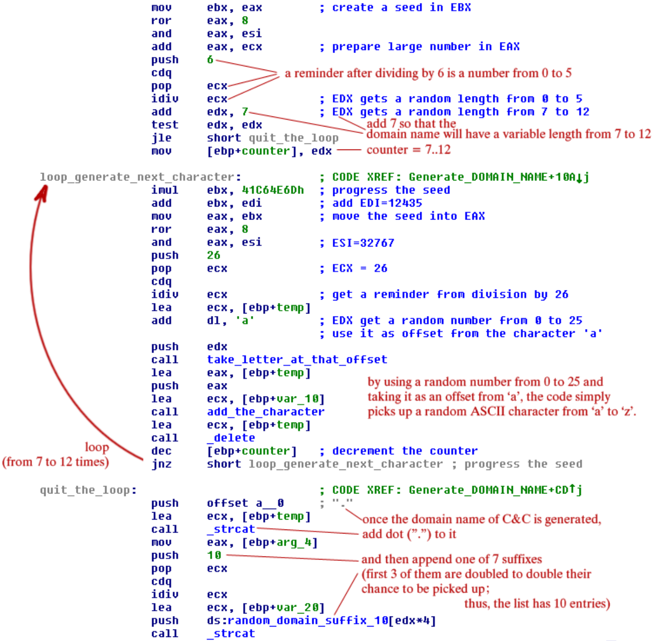
Intel's Larrabee

- Announced by Intel at SIGGRAPH 2008
- Based on the x86 architecture plus Larrabee-specific extensions
- Will also come in the form of an add-in card managed by an operating system driver
- No reference/documentation for the moment

Outline

1. GPGPU Technologies
   - CUDA
   - Stream Computing
   - OpenCL
   - Larrabee
2. How could that be used in a malware ?
3. Reverse Engineering
   - Disassembling
   - Debugging
   - Emulation
4. Packing
5. Conclusion

Quick Answer (credits: ThreatExpert.com)



```
                 mov     ebx, eax         ; create a seed in EBX
                 ror     eax, 8
                 and     eax, esi
                 add     eax, ecx         ; prepare large number in EAX
                 push    6
                 cdq
                 pop     ecx
                 idiv    ecx              ; EDX gets a random length from 0 to 5
                 add     edx, 7           ; EDX gets a random length from 7 to 12
                 test    edx, edx
                 jle     short quit_the_loop
                 mov     [ebp+counter], edx

loop_generate_next_character:             ; CODE XREF: Generate_DOMAIN_NAME+10A↓j
                 imul    ebx, 41C64E6Dh   ; progress the seed
                 add     ebx, edi         ; add EDI=12435
                 mov     eax, ebx         ; move the seed into EAX
                 ror     eax, 8
                 and     eax, esi         ; ESI=32767
                 push    26
                 pop     ecx              ; ECX = 26
                 cdq
                 idiv    ecx              ; get a reminder from division by 26
                 lea     ecx, [ebp+temp]
                 add     dl, 'a'          ; EDX get a random number from 0 to 25
                                          ; use it as offset from the character 'a'
                 push    edx
                 call    take_letter_at_that_offset
                 lea     eax, [ebp+temp]
                 push    eax
                 lea     ecx, [ebp+var_10]
                 call    add_the_character
                 lea     ecx, [ebp+temp]
                 call    _delete
                 dec     [ebp+counter]    ; decrement the counter
                 jnz     short loop_generate_next_character ; progress the seed

quit_the_loop:                            ; CODE XREF: Generate_DOMAIN_NAME+CD↑j
                 push    offset a__0      ; "."
                 lea     ecx, [ebp+temp]
                 call    _strcat
                 mov     eax, [ebp+arg_4]
                 push    10
                 pop     ecx
                 cdq
                 idiv    ecx
                 lea     ecx, [ebp+var_20]
                 push    ds:random_domain_suffix_10[edx*4]
                 call    _strcat
```

a reminder after dividing by 6 is a number from 0 to 5

add 7 so that the domain name will have a variable length from 7 to 12

counter = 7..12

loop (from 7 to 12 times)

by using a random number from 0 to 25 and taking it as an offset from 'a', the code simply picks up a random ASCII character from 'a' to 'z'.

once the domain name of C&C is generated, add dot (".") to it

and then append one of 7 suffixes (first 3 of them are doubled to double their chance to be picked up; thus, the list has 10 entries)

Algorithm Hiding

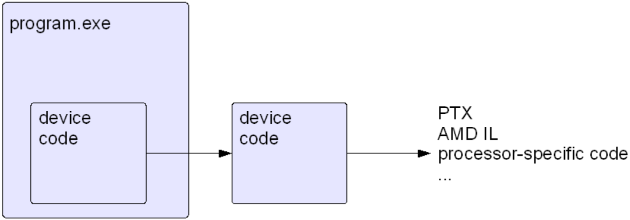
- The code on the former slide is part of the Kraken botnet
- It is the algorithm generating the list of C&C servers that the bots try to contact
- Once this list is known, the servers can be shut down and the botnet can be infiltrated
- This is the kind of algorithms that might end up being executed on GPUs

Outline

1. GPGPU Technologies
   - CUDA
   - Stream Computing
   - OpenCL
   - Larrabee
2. How could that be used in a malware ?
3. Reverse Engineering
   - Disassembling
   - Debugging
   - Emulation
4. Packing
5. Conclusion

Disassembling

- GPGPU software comes in the form of fat binaries (CUDA terminology), i.e. native executables with embedded device code
- The goal is to extract the device code and obtain a dump of the instructions

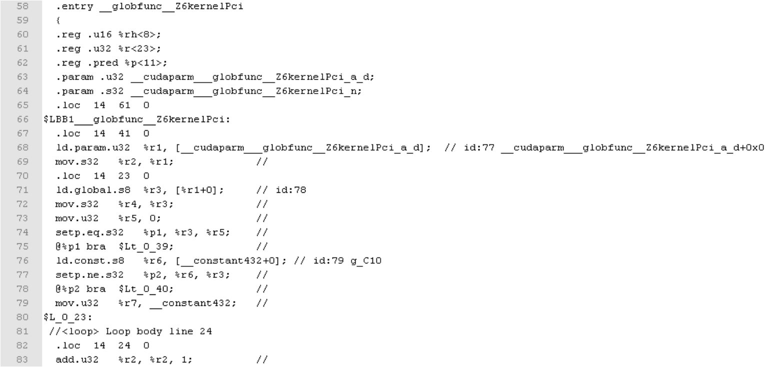


Disassembling

- Depends heavily on the underlying GPGPU technology
- Ability to recover the device-specific representation and/or the intermediate language representation
- Usually very different from x86 assembly

Disassembling

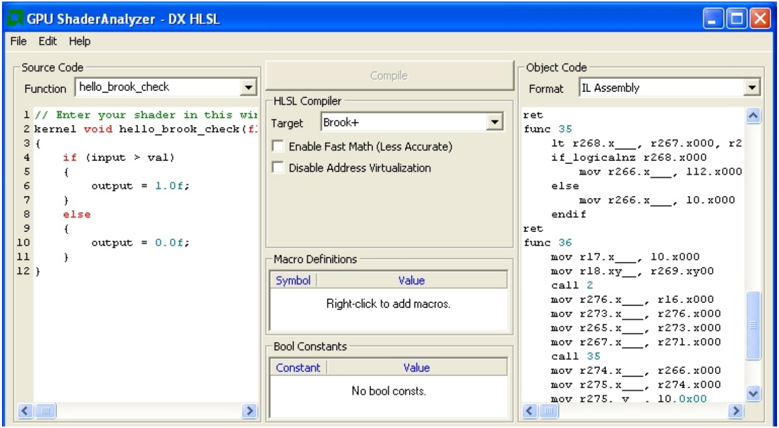
Sample PTX code:



```
  .entry __globfunc__Z6kernelPci
  {
  .reg .u16 %rh<8>;
  .reg .u32 %r<23>;
  .reg .pred %p<11>;
  .param .u32 __cudaparm___globfunc__Z6kernelPci_a_d;
  .param .s32 __cudaparm___globfunc__Z6kernelPci_n;
  .loc  14  61  0
$LBB1___globfunc__Z6kernelPci:
  .loc  14  41  0
  ld.param.u32  %r1, [__cudaparm___globfunc__Z6kernelPci_a_d];  // id:77 __cudaparm___globfunc__Z6kernelPci_a_d+0x0
  mov.s32   %r2, %r1;             //
  .loc  14  23  0
  ld.global.s8  %r3, [%r1+0];     // id:78
  mov.s32   %r4, %r3;             //
  mov.u32   %r5, 0;               //
  setp.eq.s32   %p1, %r3, %r5;    //
  @%p1 bra  $Lt_0_39;             //
  ld.const.s8   %r6, [__constant432+0]; // id:79 g_C10
  setp.ne.s32   %p2, %r6, %r3;    //
  @%p2 bra  $Lt_0_40;             //
  mov.u32   %r7, __constant432;   //
$L_0_23:
 //<loop> Loop body line 24
  .loc  14  24  0
  add.u32   %r2, %r2, 1;          //
```

Disassembling

Sample AMD IL code:



Function: hello_brook_check

```
// Enter your shader in this wi
kernel void hello_brook_check(f
{
    if (input > val)
    {
        output = 1.0f;
    }
    else
    {
        output = 0.0f;
    }
}
```

HLSL Compiler

Target: Brook+

□ Enable Fast Math (Less Accurate)

□ Disable Address Virtualization

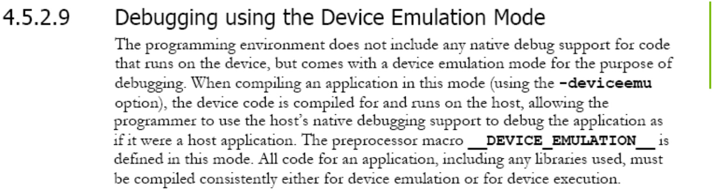
Format: IL Assembly

```
ret
func 35
    lt r268.x___, r267.x000, r2
    if_logicalnz r268.x000
        mov r266.x___, 112.x000
    else
        mov r266.x___, 10.x000
    endif
ret
func 36
    mov r17.x___, 10.x000
    mov r18.xy__, r269.xy00
    call 2
    mov r276.x___, r16.x000
    mov r273.x___, r276.x000
    mov r265.x___, r273.x000
    mov r267.x___, r271.x000
    call 35
    mov r274.x___, r266.x000
    mov r275.x___, r274.x000
    mov r275._y__, 10.0x00
```

Debugging

- Short version: GPUs do not support hardware debugging
- This means: no breakpoints, no single-stepping, no debugger-based tracing
- However, developers want to debug applications, so the answer is the emulation mode...

An excerpt of the CUDA documentation:



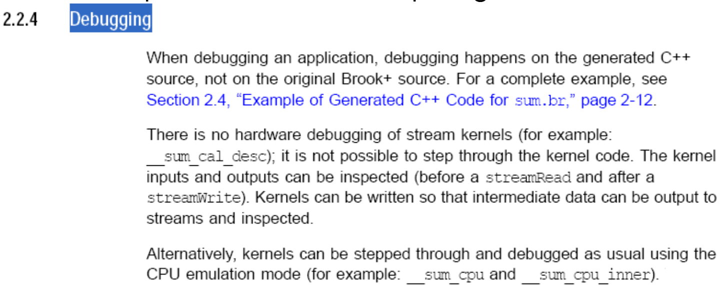
4.5.2.9 Debugging using the Device Emulation Mode

The programming environment does not include any native debug support for code that runs on the device, but comes with a device emulation mode for the purpose of debugging. When compiling an application in this mode (using the `-deviceemu` option), the device code is compiled for and runs on the host, allowing the programmer to use the host's native debugging support to debug the application as if it were a host application. The preprocessor macro `__DEVICE_EMULATION__` is defined in this mode. All code for an application, including any libraries used, must be compiled consistently either for device emulation or for device execution.

Debugging

- Short version: GPUs do not support hardware debugging
- This means: no breakpoints, no single-stepping, no debugger-based tracing
- However, developers want to debug applications, so the answer is the emulation mode...

And an excerpt of the Stream Computing documentation:



2.2.4 Debugging

When debugging an application, debugging happens on the generated C++ source, not on the original Brook+ source. For a complete example, see Section 2.4, "Example of Generated C++ Code for sum.br," page 2-12.

There is no hardware debugging of stream kernels (for example: __sum_cal_desc); it is not possible to step through the kernel code. The kernel inputs and outputs can be inspected (before a streamRead and after a streamWrite). Kernels can be written so that intermediate data can be output to streams and inspected.

Alternatively, kernels can be stepped through and debugged as usual using the CPU emulation mode (for example: __sum_cpu and __sum_cpu_inner).

Debugging

- So developers can debug their applications if they compile them with an emulation option
- This means no debugging without the source code
- But at least, we have emulation, right ?

Emulation

- Let's read again the CUDA documentation: *"When compiling an application in this mode (using the -deviceemu option), the device code is compiled for and runs on the host"*
- This means that no GPU code is produced, everything is compiled for the CPU
- Therefore, no emulation without the source code
- This is bad news for malware analysts, because having a full-software GPU emulator would allow the use of breakpoints, single-stepping and tracing (as with Bochs)

Outline

1. GPGPU Technologies
   - CUDA
   - Stream Computing
   - OpenCL
   - Larrabee
2. How could that be used in a malware ?
3. Reverse Engineering
   - Disassembling
   - Debugging
   - Emulation
4. Packing
5. Conclusion

Motivation

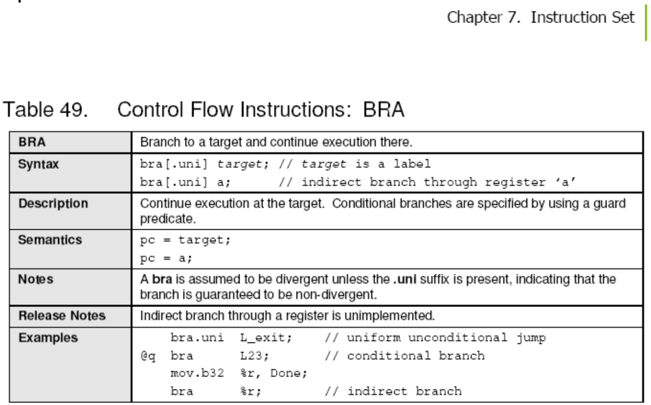
- Packing is a software protection method that generates code dynamically (turns data into code)
- To unpack a program, you generally have to set a breakpoint at the entry point of the dynamically created code or to emulate the program and match the current address with the written addresses
- No debugging in GPUs + no emulators (yet) = really strong packing

Based on the Underlying Hardware

- The lowest-level target but still hardware-independant target for execution is the intermediate language (such as PTX or AMD IL)
- To program self-modifying code, we need data-transfer instructions and control-flow instructions with the same targets
- But...

Based on the Underlying Hardware

Excerpt of the PTX documentation:



Chapter 7. Instruction Set

Table 49. Control Flow Instructions: BRA

BRA	Branch to a target and continue execution there.

Syntax	`bra[.uni] target; // target is a label`
	`bra[.uni] a;      // indirect branch through register 'a'`

Description	Continue execution at the target. Conditional branches are specified by using a guard predicate.

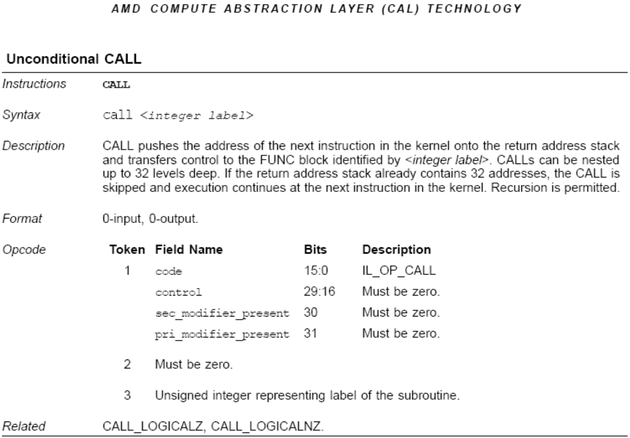
Semantics	`pc = target;`
	`pc = a;`

Notes	A **bra** is assumed to be divergent unless the **.uni** suffix is present, indicating that the branch is guaranteed to be non-divergent.
Release Notes	Indirect branch through a register is unimplemented.

Examples	`    bra.uni  L_exit;    // uniform unconditional jump`
	`@q  bra      L23;       // conditional branch`
	`    mov.b32  %r, Done;`
	`    bra      %r;        // indirect branch`

Based on the Underlying Hardware

And an excerpt of the AMD IL documentation:



AMD COMPUTE ABSTRACTION LAYER (CAL) TECHNOLOGY

Unconditional CALL

Instructions CALL

Syntax `call <integer label>`

Description CALL pushes the address of the next instruction in the kernel onto the return address stack and transfers control to the FUNC block identified by <*integer label*>. CALLs can be nested up to 32 levels deep. If the return address stack already contains 32 addresses, the CALL is skipped and execution continues at the next instruction in the kernel. Recursion is permitted.

Format 0-input, 0-output.

Opcode

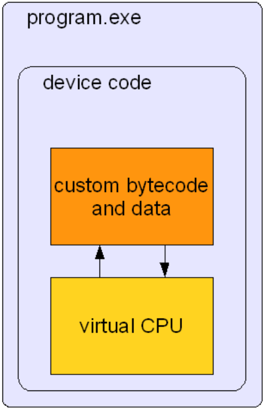
Token	Field Name	Bits	Description
1	code	15:0	IL_OP_CALL
	control	29:16	Must be zero.
	sec_modifier_present	30	Must be zero.
	pri_modifier_present	31	Must be zero.
2	Must be zero.		
3	Unsigned integer representing label of the subroutine.		

Related CALL_LOGICALZ, CALL_LOGICALNZ.

Based on a Virtual Machine

- There seems to be no natural / documented way to write self-modifying code with PTX or AMD IL
- However, even if the underlying environment does not support self-modifying code, it is still possible to develop a virtual execution environment in device code
- Since we control the virtual execution environment, everything is possible, including self-modifying code
- Not malware specific, DRM systems may use it in the future (GPU-Themida and GPU-VMProtect ?)

Based on a Virtual Machine



Outline

1. GPGPU Technologies
   - CUDA
   - Stream Computing
   - OpenCL
   - Larrabee
2. How could that be used in a malware ?
3. Reverse Engineering
   - Disassembling
   - Debugging
   - Emulation
4. Packing
5. Conclusion

Conclusion

- Current GPGPU technologies offer programmable hardware black boxes
- If one of these technologies becomes a standard, available by default, it will be used by malware and DRM
- GPU-based packers will be particularly efficient due to the lack of hardware debugging and emulators