Strategies for Protecting Intellectual Property when Using CUDA Applications on Graphics Processing Units

Xavier Bellekens
Abertay University
Dundee, United Kingdom
x.bellekens@abertay.ac.uk

Greig Paul
University of Strathclyde
Glasgow, United Kingdom
greig.paul@strath.ac.uk

Christos Tachtatzis
University of Strathclyde
Glasgow, United Kingdom
christos.tachtatzis@strath.ac.uk

James Irvine
University of Strathclyde
Glasgow, United Kingdom
j.m.irvine@strath.ac.uk

Robert Atkinson
University of Strathclyde
Glasgow, United Kingdom
robert.atkinson@strath.ac.uk

ABSTRACT

Recent advances in the massively parallel computational abilities of graphical processing units (GPUs) have increased their use for general purpose computation, as companies look to take advantage of big data processing techniques. This has given rise to the potential for malicious software targeting GPUs, which is of interest to forensic investigators examining the operation of software. The ability to carry out reverse-engineering of software is of great importance within the security and forensics fields, particularly when investigating malicious software or carrying out forensic analysis following a successful security breach. Due to the complexity of the Nvidia CUDA (Compute Unified Device Architecture) framework, it is not clear how best to approach the reverse engineering of a piece of CUDA software. We carry out a review of the different binary output formats which may be encountered from the CUDA compiler, and their implications on reverse engineering. We then demonstrate the process of carrying out disassembly of an example CUDA application, to establish the various techniques available to forensic investigators carrying out black-box disassembly and reverse engineering of CUDA binaries. We show that the Nvidia compiler, using default settings, leaks useful information. Finally, we demonstrate techniques to better protect intellectual property in CUDA algorithm implementations from reverse engineering.

CCS Concepts

•**Security and privacy** → **Software reverse engineering;** *Software and application security;*

Keywords

Reverse engineering, Intellectual Property, CUDA

SIN, July 20-22, 2016, ,
© 2016 ACM. ISBN 978-1-4503-4764-8/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2947626.2947635

1. INTRODUCTION

In its simplest form, the reverse engineering of software is the process through which the means of operation of a piece of software (for which source code is typically not available) can be discovered. The process of reverse engineering software for CPU architectures (such as x86 and x86_64) is well-documented [5] [21]. Despite this, the techniques commonly applied to CPU-based software are not necessarily optimal when dealing with CUDA, which is effectively another layer of abstraction on top of an existing computer, introducing its own nuances, which may aid the reverse engineering process.

Reverse engineering of CUDA-based GPU binaries may be necessary, for example, to investigate the extent and impact of a security breach, where suspected malicious code is executed on a GPU [12] [22]. Being able to identify the operations carried out may allow an investigator to establish the potential exposure of confidential information, or to identify what operations were carried out on data previously held on the GPU [1]. Analysis may be necessary for the purpose of creating intrusion detection system signatures [2] or the detection and classification of similar threats in future.

Reverse engineering can also be used to gain an understanding of code being executed; perhaps for the purpose of software auditing — the use of reverse engineering could ensure that software handling sensitive data is not abusing its privileges. Additionally, as GPU applications often represent a commercial investment of the originating company, reverse engineering can pose a direct threat through the theft of Intellectual Property (IP) [16].

Code intended to run on a CUDA-enabled GPU is compiled against the CUDA API on a host computer, and the resulting binary is executed on a host computer. The standard ELF-based binary is executed on the host CPU like any other software, and carries out initialisation of the graphics processor before loading the desired CUDA code on to GPU, where the code is ultimately executed. CUDA code is therefore not executed directly on the host CPU; rather it is transferred into GPU memory via the host initialisation routine. The CUDA code is subsequently executed following a call from the host to the CUDA driver API [23]. The presence of a host-executable ELF binary is of significance to the reverse engineering process as the operation of this host binary can be investigated using standard techniques, yet this analysis alone will not yield any information as to the code executed on the GPU.

Given that GPUs come into contact with potentially sensitive data being processed, they are a clear and attractive target for malicious software [22] [19]. For this reason, writers of malicious software may wish to prevent their code from being understood, and those using GPUs for data processing may wish to understand the actions a piece of software carried on their GPU.

In the case of forensic investigation, the ability to analyse GPU-based memory may yield important clues as to the data which was processed by the GPU and therefore what data may have been accessed by an attacker, as a result of data remanence on GPUs [1]. This analysis, however, requires a forensic investigator to know specific details of memory allocation of the CUDA binary, which would need to be obtained through reverse engineering of the target CUDA software. A clear understanding of the various CUDA binaries likely to be encountered and means of understanding their relation to the code being executed, is therefore highly beneficial for an investigator, and forms the basis of our contributions.

This paper makes four contributions. Firstly, it demonstrates that the default Nvidia CUDA compiler settings significantly aid in reverse engineering. Secondly, strategies are presented to improve protection of intellectual property within CUDA software, and reduce the ease of reverse engineering. Thirdly, static and dynamic analysis of CUDA binaries is explored, with exploration of the tools available. Finally, the reverse engineering strategies presented are demonstrated as effective against various CUDA binaries.

2. REVERSE ENGINEERING AND RELATION TO CUDA

The field of reverse engineering is best split into two categories — those of static and dynamic analysis. In static analysis, the target binary is not actually executed; instead, it is loaded into a disassembler. The disassembler parses the compiled executable binary, mapping machine code operations back to the somewhat more human-readable assembly listing format. This process is carried out on the compiled program code, since the source code is not available. Various static analysis tools are available from the most simple GNU tools with origins from original Unix operating systems, such as the `strings` utility which looks for readable text strings contained within a file, through to specialist Nvidia tools to extract information from CUDA binaries. These are discussed in greater detail in Sections 5 and 6.

In contrast, dynamic reverse engineering involves the execution of the code, typically on a virtualised or debuggable physical environment [10]. Through the use of a debugger, which permits the flow of the program code to be monitored, interrupted and altered, it is possible to gain an insight into the operation of the code, and a clearer understanding as to the areas of interest within a program. For example, the system memory may be monitored for the presence of a particular value of interest, triggering a breakpoint when detected, thus highlighting that the code is handling a particular value. It can also be used to investigate the operation of a given section of complex code, through sequential execution with various selected input values. Debugging can also be used to identify function or system calls of interest, such as those which are defined within the CUDA API libraries, or calls to the CUDA driver.

Within the context of analysis of CUDA code being executed on a GPU, it is important to note the distinction between host code, which is executed on the host platform CPU, like any other piece of software, and device code, which is executed by the CUDA driver on the GPU. While host code can be reverse engineered using standard techniques [9], there is currently very limited research into the background information necessary to investigate the CUDA-based code [22] [20].

Having the ability to investigate the operation of CUDA-based software is of great importance, specifically given the well-documented use of GPUs for the unauthorised processing of potentially confidential data [1], as well as for use in compression and cryptography [15]. With sensitive data routinely being exposed to GPUs, the ability for forensic investigators to respond to security incidents involving the potential compromise of these systems and rapidly investigate the precise nature of the suspect code involved, is critical to presenting an effective incident response.

3. OVERVIEW OF CUDA ARCHITECTURE

The Compute Unified Device Architecture framework [17] allows developers to take advantage of the massively parallel processing capabilities of graphics processors, allowing GPUs to become widely available general purpose computing devices. The CUDA framework largely extends the functionality of the C99/C++ languages, allowing developers to take full advantage of the parallel nature of the hardware, through the use of a flexible abstraction model.

CUDA applications are compiled by `nvcc` (the Nvidia Cuda Compiler). The compiler takes a set of source files as input, and generates a variety of compiled output formats, allowing backward and forward compatibility between devices. The standard output format from the `nvcc` compiler is an ELF-based host executable, which is executed like any standard program on the host CPU. This program then initialises the GPU via the CUDART library, and loads the relevant CUDA code onto the GPU. The ELF binary is a container, holding a variety of different formats of program code.

The GPU code present in the CPU-executed ELF binary is known as a *kernel*. When invoked, the kernel first allocates the required memory on the GPU, and loads the desired data onto the device for processing. After processing, the resulting output is transferred back to the host CPU memory and the GPU memory is released for use by other applications [3].

4. CUDA BINARIES

While a regular piece of software to be executed on the host CPU would typically be translated to machine code, there are a number of different output formats that CUDA code can take. All of these output format can be bundled in the ELF binary. Some of these make the process of reverse engineering the compiled code considerably easier than others, yet may still be found in compiled binaries.

The highest level of abstraction is the original CUDA C/C++ source code. It is highly unlikely, however, that source code for a given binary would be available to a forensic investigator attempting to ascertain what was being done by a piece of software on a GPU. Nonetheless, if open-source

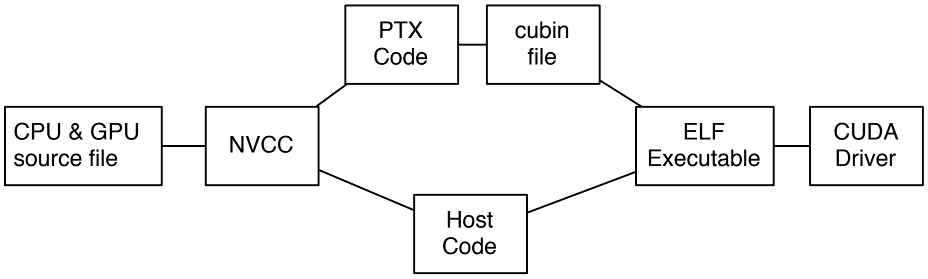


Figure 1: Compilation Process

code was being used (and an investigator could determine this), it may be possible to acquire the source code from the original authors and inspect it to understand the operation of that code.

The CUDA source code of a given piece of CUDA software is not distributed as part of the executable binary. Like with regular programming languages for host CPUs, the source code undergoes compilation to produce an executable binary for the target instruction set. Due to the rapid development and improvements in GPU technology and their associated capabilities, CUDA GPUs do not all support the same baseline instruction set as is the case with an x86-based CPU, for example. Where almost all general purpose desktop CPUs support the 8086 instruction set with full backward compatibility (from 1978), this is not the case with Nvidia CUDA GPUs.

Compiled CUDA binary code is often referred to as a Cubin, a contraction of CUDA and binary, which contains executable machine code, designed to run on a given target architecture. As a result of the heavily architecture-dependent nature of Cubin executables, the Nvidia CUDA `nvcc` compiler compiles CUDA source code into a platform-independent intermediary language, known as PTX. PTX is laid out in an assembly-like structure (using basic primitive instructions for all operations), while still remaining GPU-agnostic (allowing for the same code to be executed on different GPUs) [18]. An example of PTX code is shown in Section 5. It is produced by the `nvcc` compiler, which translates CUDA code into PTX (rather than directly to assembly, as a conventional CPU compiler would). The CUDA graphics driver can process the resulting PTX, producing native binary code (a cubin), capable of being executed on the GPU [6] [4]. Cubin files are composed of PTX (Parallel Thread eXecution) code, and SASSM (Streaming ASSeMbly) instructions as shown in Figure 1.

If a host ELF binary contains PTX intermediary code, a suitable CUDA binary for any supported GPU architecture can be compiled at runtime (including GPUs which were not released at the time of writing the code), since the PTX code is platform-agnostic and can be targeted at the detected GPU architecture based on the register mapping data contained within the CUDA driver. By including PTX code in the output binary, the compiled CUDA host ELF executable will be as generic as possible, and will enjoy a degree of forward compatibility with future GPUs [23]. Given the nature of the PTX format, however, its presence will significantly aid the disassembly and reverse engineering processes, since it yields highly readable and generic intermediate-language code somewhat similar in nature to Android's intermediary Dalvik code [11], that can be understood with relative ease.

SASSM is used as a definition of the instruction set implemented by the GPU. Each major GPU architecture (Tesla, Fermi, Kepler) has its own instruction set [23]. This allows new GPU architectures to change their underlying technical design, to improve performance without need for consideration of backwards compatibility. By combining the appropriate target specific register definitions with platform-independent PTX code, the GPU driver can produce an executable SASSM re-bundled into a CUDA binary (Cubin), designed for execution on the correct GPU architecture.

As such, from the perspective of a forensic investigator wishing to carry out an investigation of code running on a GPU, they are likely to encounter one of two scenarios. The first (most likely) scenario is that the code running on the GPU was executed using a host binary containing PTX code for the algorithm in question. Since the host binary contains the PTX code, it is relatively straightforward to extract the PTX section of the binary (using the tools discussed in Section 5) and use it to determine the operation of the CUDA code in question. The second and less likely scenario is that the code being executed on the GPU has the PTX section stripped, perhaps in order to hinder inspection or reverse engineering of the code. In this case, the platform-specific compiled Cubin code must therefore be disassembled, to yield a listing of the machine code being executed.

5. STATIC ANALYSIS OF CUDA ELF BINARIES

The SANS institute recommends static analysis as the first step when performing reverse engineering [7]. Different strategies can be used against Cubins to reveal the operations carried out by the software. The reverse engineering was performed against simple open-source applications, to demonstrate the information which could be recovered from the host ELF executable.

The first sample application used to demonstrate the reverse engineering process is a simple hash brute-forcing application (which is a task that GPUs are commonly used for, given their highly parallel nature). The application accepts two inputs — one MD4/5 hash, and one dictionary containing plain text words, both of which are transferred to the GPU by the host binary. A comparison is then made between the computed MD4/5 hashes of each word contained in the dictionary and the original MD4/5 hash and if a match

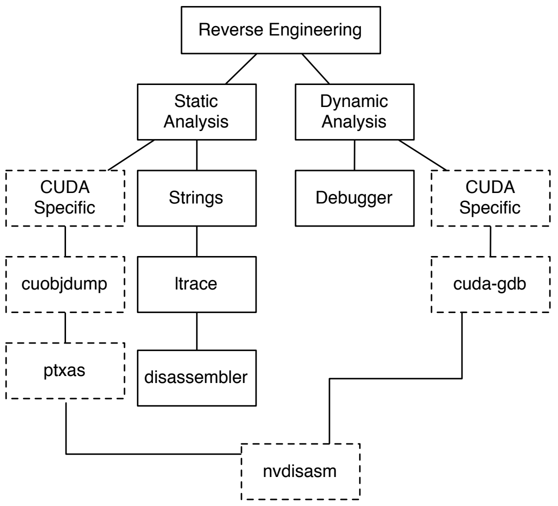


Figure 2: Reverse Engineering Tools

```
__cudaparm__Z5bruteforce_dict_a
__cudaparm__Z5bruteforce_md5C_b
__cudaparm__Z5bruteforce2_dict_a
__cudaparm__Z5bruteforce2_md4_b
```

Figure 3: Recovering all Function Instances and Arguments

is found, it is returned to the host CPU.

Figure 2 shows a selection of the key tools available on Unix systems, as well as some CUDA specific tools, which are useful for reverse engineering purposes. These tools are not all specifically designed for the purpose of reverse engineering — many are standard base utilities, which can be used to retrieve essential information from binaries. Fundamentally, for someone attempting to reverse engineer a CUDA application, there is considerable potential for side-information to be identified simply from these basic standard utilities.

For example, the standard GNU `strings` utility, executed against the host ELF file produced by `nvcc` allows the forensic investigator to retrieve the name of every function in the code. The `strings` tool is designed to highlight all printable text strings of 4 or more characters in length. This is naturally useful if original strings (such as variable or function names) were not stripped out of the binary, or modified. We found this to be the default behaviour of the `nvcc` compiler, meaning that there is a good chance that useful strings may be found from cursory inspection of the host ELF in this way.

Both CUDA-specific tools and the common CPU reverse engineering tools can be used against the CUDA-produced ELF binary. Figure 3 shows the name of the functions used for a simple MD4 and MD5 brute-force application. Two functions have been found (bruteforce, bruteforce2) as well as their required arguments, the dictionary and the md4-5 hash — `dict_a` being the first argument and the `md4-5_b` being the second argument.

Being able to see the names of functions, as well as their

```
__global__
void bruteforce(char dict, char md5){
...
}
```

Figure 4: Original Function Declaration

```
bruteforce
leftSalt
rightSalt
dictionary
password
```

Figure 5: Name of the Constant Variables in the Software.

incoming parameter definitions, can be useful when carrying out reverse engineering, as it offers a rapid way to gain familiarity with the layout of the code in question. Additionally, presuming the developer of the CUDA software was not taking active steps to avoid this analysis (such as deliberately naming functions in a meaningless manner), there may be considerable side information about the operation of the program able to be determined simply from the naming of functions and parameters.

Figure 4 shows the original function declaration in the source code of our example application. The function is named "bruteforce" and receives two arguments, "dict" and "md5" as shown in Figure 3. Names of the constant variables can also be discovered in the executable, using the `strings` command. Figure 5 shows the constant variable names used in the application. This behaviour is unlike traditionally compiled C applications and may be useful for an investigator wishing to gain a quick understanding of the operation of the code. This will aid in the planning of further analysis.

Furthermore, using the `strings` tool, the forensic investigator can find cudaError comments embedded in the code to analyse the overall behaviour of the application.

Figure 6 shows two cudaErrors defined by the developer of the original code, as well as one cudaError type defined via the CUDA framework. These custom error messages can be valuable to understand the algorithmic flow. By using CUDA specific tools, the forensic investigator can retrieve useful information initially intended for the programmer.

5.1 cuobjdump

The `cuobjdump` tool allows the forensic investigator to dump information directly from the the binary, such as the `.section` assembly directives, which can yield valuable information about the different variables, functions and arguments, as well as information about the memory layout of function arguments and the type of storage used for each

```
checkCUDAError("Wrong Memory Allocation")
checkCUDAError("UnAllocated Variable")
checkCUDAError(cudaError_t cuError)
```

Figure 6: CudaError Comments Embedded in the Code.

PTX	SASSM
1 ld.param.u64 %rd1, [_Z5VecAddPcPi_param_0];	1 MOV R1, c[0x0][0x44];
2 ld.param.u64 %rd2, [_Z5VecAddPcPi_param_1];	2 S2R R0, SR_TID.X;
3 cvta.to.global.u64 %rd3, %rd1;	3 MOV32I R3, 0x4;
4 cvta.to.global.u64 %rd4, %rd2;	4 IMAD.U32.U32 R6.CC, R0, R3,c[0x0][0x148];
5 mov.u32 %r1, %tid.x;	5 IMAD.U32.U32.HI.X R7, R0, R3,c[0x0][0x14c];
6 cvt.u64.u32 %rd5, %r1;	6 IADD R4.CC, R0, c[0x0][0x140];
7 mul.wide.u32 %rd6, %r1, 4;	7 IADD.X R5, RZ, c[0x0][0x144];
8 add.s64 %rd7, %rd4, %rd6;	8 LD.E R0, [R6];
9 ld.global.u32 %r2, [%rd7];	9 LD.E.U8 R3, [R4];
10 add.s64 %rd8, %rd3, %rd5;	10 IADD R0, R3, R0;
11 ld.global.u8 %r3, [%rd8];	11 ST.E.U8 [R4], R0;
12 add.s32 %r4, %r3, %r2;	12 EXIT;
13 st.global.u8 [%rd8], %r4;	13 BRA 0x70;
14 ret;	14 NOP;

Figure 7: Comparing PTX code and SASSM code

variable. It is also possible to dump the PTX code, as well as the SASSM code, and a standalone Cubin binary.

A second code sample is shown in Figure 7, to highlight the difference between the PTX code on the right and the SASSM code on the left. This code demonstrates a simple vector addition between two variables.

The PTX code shows the name of the function and the number of parameters received (line 1, 2). It shows that both are stored in global memory (line 3, 4). The vector addition is then performed. Similar instructions can be seen for the SASSM code, however, as PTX is a higher-level language, it is easier to understand, and infer the kernel C source code, as discussed by Dong et al. [8]. In particular, PTX contains variable names and function names, conveying semantic information, unlike SASSM which is directly using registers.

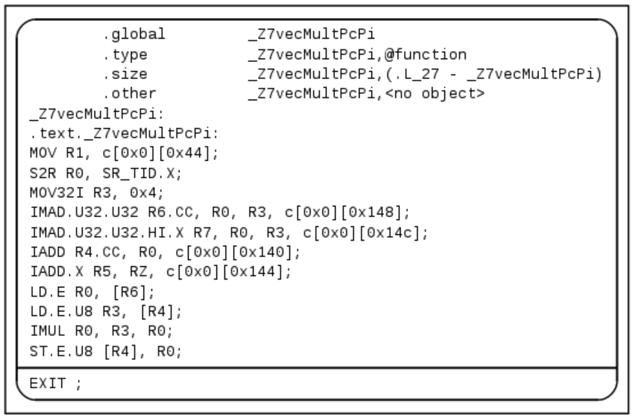
5.2 ptxas

The `ptxas` tool compiles PTX code into GPU-specific micro-code. After retrieving the PTX code from the ELF binary, the forensic investigator can use the `ptxas` tool to create an object file of the CUDA functions, if they wish to carry out further dynamic analysis, and actually execute the code in a controlled environment.

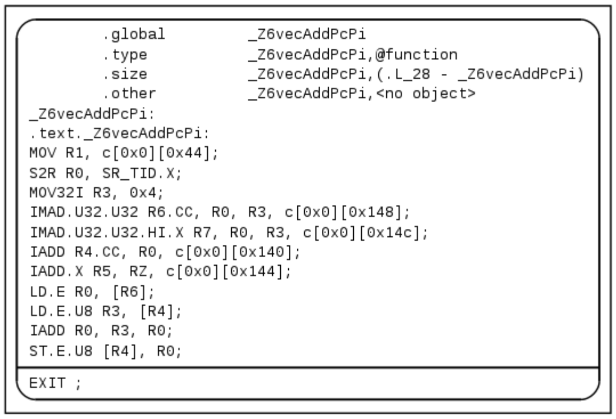
Using the PTX code, it is also possible to add debug information to the object file, as well as line number information, to assist during later stages of the disassembly process.

5.3 nvdisasm

`Nvdisasm` is an hybrid static and dynamic analysis tool, which allows a forensic investigator to extract information from the standalone CUDA binary (Cubin) and present this information in a readable format. As such, it is possible to create a flowchart of the different function calls used in the software, as well as their relation to each other as shown in Figure 8. The tool is also able to show register usage alongside each executed instruction, and can therefore be considered a form of dynamic analysis tool, since it effectively executes the code, in order to establish which registers would be used to store values during execution. This is useful if attempting to establish the contents of a register at a given point, perhaps due to later usage of that register elsewhere in the PTX or disassembled code. In the case of a multi-tenant computing situation, where GPUs are shared between several independent users of a system such as in platform-as-a-service situations, this could be used to carry out pre-execution behavioural checks against CUDA software, to attempt to identify malicious behaviours, such as attempting to dump the contents of all GPU registers, or to allocate the full extent of available GPU memory in a single array. This could be attempts to exploit data remanence of GPUs to extract data processed by a previous user of the shared GPU [1].



```
        .global         _Z7vecMultPcPi
        .type           _Z7vecMultPcPi,@function
        .size           _Z7vecMultPcPi,(.L_27 - _Z7vecMultPcPi)
        .other          _Z7vecMultPcPi,<no object>
_Z7vecMultPcPi:
.text._Z7vecMultPcPi:
MOV R1, c[0x0][0x44];
S2R R0, SR_TID.X;
MOV32I R3, 0x4;
IMAD.U32.U32 R6.CC, R0, R3, c[0x0][0x148];
IMAD.U32.U32.HI.X R7, R0, R3, c[0x0][0x14c];
IADD R4.CC, R0, c[0x0][0x140];
IADD.X R5, RZ, c[0x0][0x144];
LD.E R0, [R6];
LD.E.U8 R3, [R4];
IMUL R0, R3, R0;
ST.E.U8 [R4], R0;
EXIT ;
```



```
        .global         _Z6vecAddPcPi
        .type           _Z6vecAddPcPi,@function
        .size           _Z6vecAddPcPi,(.L_28 - _Z6vecAddPcPi)
        .other          _Z6vecAddPcPi,<no object>
_Z6vecAddPcPi:
.text._Z6vecAddPcPi:
MOV R1, c[0x0][0x44];
S2R R0, SR_TID.X;
MOV32I R3, 0x4;
IMAD.U32.U32 R6.CC, R0, R3, c[0x0][0x148];
IMAD.U32.U32.HI.X R7, R0, R3, c[0x0][0x14c];
IADD R4.CC, R0, c[0x0][0x140];
IADD.X R5, RZ, c[0x0][0x144];
LD.E R0, [R6];
LD.E.U8 R3, [R4];
IADD R0, R3, R0;
ST.E.U8 [R4], R0;
EXIT ;
```

Figure 8: Representation of the different functions

6. DYNAMIC ANALYSIS OF CUDA BINARIES

Dynamic analysis is the second step of our proposed reverse engineering process. This analysis makes use of information previously retrieved from static analysis, such as the function flowchart, the PTX code and the object files generated. The CUDA framework comes with its own debugger, allowing a forensic investigator to run the host CPU code in a pseudo-virtualised environment.

6.1 cuda-gdb

`Cuda-gdb` is the official CUDA debugger. It allows programmers to run their algorithm on the host system, while being able to use a set of debug instructions to pause and step through code, analyse memory requirements, variable values and thread positions. `Cuda-gdb` is based upon the GNU gdb utility, which is a standard x86 debugger.

While running code through the `cuda-gdb` debugger, the investigator is able to place breakpoints within the compiled SASSM code, which is the machine-code that is actually executed on the GPU, in order to better understand the behaviour of the software, as shown in Figure 9. The

Table 1: Reverse Engineering Strategy and Extraction Result Summary

	ELF Binary				Tools			
	ELF	CUBIN	PTX	SASSM	cuobjdump	ptxas	nvdisasm	cuda-gdb
Variables	Yes	Yes	No	No	No	No	No	No
Functions Name	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Functions Arguments	Yes	Yes	No	No	No	No	No	No
CUDA Errors	Yes	Yes	No	No	No	No	No	No
PTX	Yes	Yes	N/A	No	Yes	No	No	No
SASSM	Yes	Yes	No	N/A	Yes	No	Yes	Yes
CUBIN	Yes	N/A	No	No	Yes	No	Yes	No
Code Logic	No	No	Yes	Yes	No	Yes	Yes	Yes

```
1 (cuda-gdb) b _Z6vecAddPiS_S_
2 Breakpoint 1 at 0x4026ff
3 (cuda-gdb) r
4 (cuda-gdb) cuda kernel block thread
5 kernel 1, block(1,0,0), thread(1024,0,0)
```

Figure 9: Cuda-gdb primitives to analyse the behaviour of the code

```
./vectorAdd
Vector Addition
Using Device 0: "Tesla K20m"
Module Path <./vectorAdd_kernel64.ptx>
loading module: <./vectorAdd_kernel64.ptx>
```

Figure 10: Just In Time Compilation

first line places a breakpoint on the GPU kernel "VecAdd" and the third line then executes the main CPU code, thus launching the kernel on the GPU. The breakpoint is then triggered when the kernel loads, allowing for further debug commands to be executed, such as inspecting variable values or stepping through code line-by-line. For example, the investigator can request the debugger provide information on the state of the kernel, such as the number of threads (line 5), when the breakpoint is encountered, the code execution temporarily pauses.

Furthermore, the debugger can be used to ascertain more detailed information on nuances of the flow of the software, and to complete any missing information that static analysis did not reveal. The PTX code obtained during static analysis can also be used to extend dynamic analysis, by making small adjustments to its assembler-like code, in order to force a specific set of operations to be carried out. The Just-In-Time (JIT) compilation capabilities of the framework are then used to create and execute the PTX code rapidly on the GPU, allowing for rapid modifications and testing cycles to be completed.

As shown in Figure 10, by loading both the object file and PTX code at runtime, the forensic investigator can modify the behaviour of the PTX code before it is JIT-compiled and executed, therefore allowing for further dynamic analysis of the code. By making small alterations to the instructions in the PTX code as needed, the JIT compiler then produces a new object file, which is executed, allowing for the rapid modification and testing of modified versions of the CUDA code. This allows different portions of the code to be examined in isolation, and can reduce the time needed for dynamic analysis by allowing unneeded portions to be "short-circuited" out in the PTX, thus allowing an investigator to focus only on areas of interest. Being able to rapidly alter and execute modified code is a significant advantage to anyone attempting to reverse engineer the software, although this is only possible if a PTX section is present in the binary. We found this to be be the default behaviour of the nvcc compiler, meaning it is likely that CUDA binaries encountered elsewhere will contain this section, thus aiding dynamic analysis.

The results achieved by the reverse engineering strategy, using the different types of outputs produced by the different formats generated by the nvcc compiler are summarised in Table 1 where "Yes" represents the possibility of retrieving data, "No" the impossibility of retrieving data and "N/A" indicates a given test was not applicable. The results demonstrate the possibilities of reverse engineering for forensic investigators and highlight possible intellectual property concerns regarding the format and informations yielded by the different binaries. Perhaps most significantly, these findings highlight that more information about a given CUDA program can be obtained from direct analysis of the binaries, which is not readily exposed using the NVidia tools. It is therefore important for developers to note that a binary which reveals no valuable information when inspected with the NVidia tools may still contain such information, accessible directly through the techniques we have discussed.

7. RECOMMENDATIONS TO PROTECT INTELLECTUAL PROPERTY

It should be noted that as with all security measures which attempt to prevent the extraction of intellectual property from compiled software code, the code in question must ultimately be executed on the GPU, meaning that attempts to make reverse engineering and disassembly more difficult are ultimately a form of security through obscurity. It is

Standard Compilation (**Part A**)

```
$ nvcc vector.cu -arch=compute_20 -code=compute_20,sm_20
$ /usr/local/cuda-6.5/bin/cuobjdump -ptx ~/ReverseEngineering/ReverseEngineering/a.out

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin ptx code:
================
arch = sm_20
code version = [3,2]
producer = cuda
host = linux
compile_size = 64bit
compressed
identifier = vector.cu

.version 3.2
.target sm_20
.address_size 64

.visible .entry _Z10vector_addPiS_S_(
.param .u64 _Z10vector_addPiS_S__param_0,
.param .u64 _Z10vector_addPiS_S__param_1,
.param .u64 _Z10vector_addPiS_S__param_2,
[...]
```

Fat Binary Compilation (**Part B**)

```
$ nvcc vector.cu -gencode arch=compute_20,\"code=sm_20\" -gencode arch=compute_30,\"code=sm_30\"
$ /usr/local/cuda-6.5/bin/cuobjdump -ptx ~/ReverseEngineering/ReverseEngineering/a.out

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = vector.cu

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = vector.cu
```

Figure 11: Comparison of Standard and Fat Compilation Process.

clear, however, that it is possible to produce binaries which are less easily reverse engineered, thus increasing the time and expertise needed to carry out analysis and gain an understanding of the binary in question.

In the field of CPU-executed code, on standard operating system platforms, obfuscated compilers exist [13], which are designed to produce unintuitive and difficult to reverse engineer binaries, which feature extraneous complexity in their compiled code. These complexities may have an impact on performance, if there is an increase in the number of instructions executed by the processor. For massively parallel processing (such as that carried out by GPU), performance of the code in question is significant, with the same code being executed in parallel across many hundreds of cores simultaneously. Any increase in the number of instructions carried out by the streaming multiprocessor would result in a significant reduction in performance across all cores. For a higher number of simultaneous threads being executed, the number of wasted clock cycles added by the obfuscation would increase, since each stream processor runs the same code and it would ultimately lead to more divergence of the threads.

While it would be possible to wrap the CUDA host ELF binary to make it more difficult to carry out static analysis, it is worth considering that ultimately the GPU must have the full Cubin binary uploaded to it. Such obfuscation, while not impacting on performance of the calculations, would be relatively easily overcome by using dynamic analysis to extract and store the code sent to the GPU driver. By using the `cuda-gdb` debugger to carry out dynamic analysis, it would be possible to step through execution of the host binary until it had suitably decrypted or de-obfuscated the Cubin or PTX code, and then extract it for direct analysis.

We therefore recommend that, as per our demonstration of the reverse engineering process of a CUDA binary, a developer may attempt to hinder reverse engineering through the translation of variable and function names into placeholder ones, thus giving less side information to a reverse engineer. This strategy is similar to that used in Java by tools such as `dexguard` and `proguard` [14], yet proves effective in slowing down static analysis of the code in question. Using this technique would offer some protection of both the Cubin and PTX code from simple static analysis, by posing a stumbling block to easy visual inspection of the code. Alternatively, a tool like `strip` could potentially be modified to support CUDA, allowing for the removal of readable strings from compiled code as demonstrated in Figure 12, `Z53456789` represents the obfuscated function name, and `234` represents the first argument name of the function.

```
__cudaparm__Z53456789_234_a
__cudaparm__Z53456789_236_b
```

Figure 12: Obfuscated Functions and Arguments Names

Additionally, if the CUDA code is able to be targeted to a single GPU architecture, and does not need forward compatibility the PTX code present in the CUDA binary can be removed when the binary code for the architecture is present and we found that the resulting modified binary still executed correctly on the target platform, despite the PTX listing being removed. If it is necessary for code to be executable on many different GPUs, a "fat" binary can be created for each GPU architecture and platform such as shown in Figure 11. The first command executed in Figure 11 (Part A) shows the process to compile a simple CUDA code file, using the default compiler settings. The second command shows the retrieval of the PTX code from the binary file resulting of the compilation, using `cuobjdump`. As expected, it is possible to retrieve the PTX code from the binary (Bottom of Part A). During the “fat” binary compilation, as shown in Figure 11 (Part B), the first line shows the command required to compile a “fat” binary using nvcc. The second line shows that, upon inspection by `cuobjdump`, the ELF binary no longer contains a PTX section. While this dramatically increases the size of the ELF binary, as it contains the SASSM code for each target architecture, it hinders reverse engineering by preventing the use of the Just-In-Time technique described in Section 6.

8. CONCLUSION

The process of reverse engineering CUDA executables is less popular in the research and scientific community compared to their CPU counterparts. In this work we have demonstrated that CUDA compiler lacks maturity, and that generated ELF binaries include high-level assembly PTX code, which can be exploited for reverse engineering or malicious purposes. We have shown that PTX code can be modified and compiled in a Just-In-Time fashion, allowing for relatively straightforward dynamic analysis. Combined with readily available Nvidia CUDA debugging tools, we have highlighted that both static and dynamic analysis of CUDA binaries can be carried out. To attempt to protect intellectual property and hinder reverse engineering of CUDA code, we recommend that developers do not distribute platform-independent binaries and instead distribute a set of GPU-architecture specific binaries with the PTX section stripped, considering the ease with which it can be used for reverse engineering. We have identified that the current default settings of the `nvcc` compiler produce binaries which yield significant side-information as to function definitions and parameters, which may prove useful to those attempting to reverse-engineer the code. We have highlighted the implications of these findings and shown that considerably more information can be extracted from binaries than is made available via the Nvidia debugging tools. We finally proposed and demonstrated two potential counter-measures to attempt to hinder analysis of CUDA binaries.

9. REFERENCES

[1] X. J. A. Bellekens, G. Paul, J. Irvine, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham. Data remanence and digital forensic investigation for CUDA graphics processing units. In *Proceedings of the 1ST IEEE/IFIP Workshop on Security for Emerging Distributed Network Technologies*, DISSECT 2015. IEEE, 2015.

[2] X. J. A. Bellekens, C. Tachtatzis, R. C. Atkinson, C. Renfrew, and T. Kirkham. A highly-efficient memory-compression scheme for GPU-accelerated intrusion detection systems. In *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN ’14, pages 302:302–302:309, New York, NY, USA, 2014. ACM.

[3] S. Breß, S. Kiltz, and M. Schäler. Forensics on GPU coprocessing in databases - research challenges, first experiments, and countermeasures. In *Datenbanksysteme für Business, Technologie und Web (BTW), - Workshopband, 15. Fachtagung des GI-Fachbereichs ”Datenbanken und Informationssysteme” (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 115–129, 2013.

[4] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.

[5] E. Chikofsky and I. Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.

[6] S. Cook. *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*. Applications of GPU computing series. Morgan Kaufmann, 2013.

[7] D. Distler. SANS institute - malware analysis: An introduction.

[8] Q. Dong, T. Li, S. Zhang, X. Jiao, and J. Leng. Ptx2kernel: Converting ptx code into compilable kernels. 2015.

[9] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2011.

[10] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. gVirtuS: A GPGPU transparent virtualization component.

[11] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In M. Huth, N. Asokan, S. Čapkun, I. Flechais, and L. Coles-Kemp, editors, *Trust and Trustworthy Computing*, volume 7904 of *Lecture Notes in Computer Science*, pages 169–186. Springer Berlin Heidelberg, 2013.

[12] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You can type, but you can’t hide: A stealthy GPU-based keylogger. *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[13] D. Low. Protecting Java code via code obfuscation. *Crossroads*, 4(3):21–23, Apr. 1998.

[14] K. Makan and S. Alexander-Bown. *Android Security Cookbook*. Packt Publishing, 2013.

[15] S. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68, Nov 2007.

[16] G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[17] Nvidia. CUDA toolkit documentation v7.0.

[18] Nvidia. Using inline PTX assembly in CUDA.

[19] D. Reynaud. GPU powered malware. In *Ruxcon*, Sydney, Australia, 11 2008.

[20] P. Stewin, J.-P. Seifert, and C. Mulliner. Poster: Towards detecting DMA malware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS ’11, pages 857–860, New York, NY, USA, 2011. ACM.

[21] T. Systä and U. Tamperensis. Static and dynamic reverse engineering techniques for Java software systems. 2000.

[22] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. GPU-assisted malware. *International Journal of Information Security*, pages 1–9, 2010.

[23] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.