6-29-2016

Enforcing Security Policies On GPU Computing Through The Use Of Aspect-Oriented Programming Techniques

Bader Albassam
University of South Florida, bader@mail.usf.edu

Enforcing Security Policies On GPU Computing

Through The Use Of Aspect-Oriented Programming Techniques

by

Bader AlBassam

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Xinming Ou, Ph.D.
Yicheng Tu, Ph.D.

Date of Approval:
June 20, 2016

Keywords: Programming Languages, Enforceability Theory, Distributed Computing, Parallel Computing, CUDA

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis presents a new security policy enforcer designed for securing parallel computation on CUDA GPUs. We show how the very features that make a GPGPU desirable have already been utilized in existing exploits, fortifying the need for security protections on a GPGPU. An aspect weaver was designed for CUDA with the goal of utilizing aspect-oriented programming for security policy enforcement. Empirical testing verified the ability of our aspect weaver to enforce various policies. Furthermore, a performance analysis was performed to demonstrate that using this policy enforcer provides no significant performance impact over manual insertion of policy code. Finally, future research goals are presented through a plan of work. We hope that this thesis will provide for long term research goals to guide the field of GPU security.

CHAPTER 1

INTRODUCTION AND MOTIVATION

General Purpose Graphical Processing Unit (GPGPU) computation is a method of computation in which a *Graphical Processing Unit* (GPU) handles the processing that would traditionally be done on a CPU. To properly explain software written for GPGPU, the architectural differences between traditional CPU based computing and GPU based computing needs to be explained. This section will serve as a short introduction, where we will be comparing the different levels of parallelism between code written for CPUs and GPUs.

A traditional single core CPU is classified as a Single Instruction Single Device (SISD) device under Flynn's taxonomy [11], which is a classification of computer architectures. In the SISD classification of architectures, a CPU has a single processing stream for a single stream of instructions running linearly on the processing core of the CPU. In order to manipulate a large amount of data, a programmer can only use a loop or recursion. This process is done through iterating along the entire data set, running the same instructions, repeatedly.

There is no hardware level parallelism implemented with the SISD classification of computer architectures. A programmer is limited to running sequential code only if they wish to fully utilize the underlying hardware of a SISD device. Thread level and instruction level parallelism can be utilized with single core CPU's to allow for multiple processes to run simultaneously. It is important to note that this level of parallelism is performed through a layer of software-based scheduling and in itself is not a hardware feature.

Modern CPUs have multiple cores running alongside each other, allowing a degree of hardware level parallelism to the programmer. These processor cores share memory but are capable of running on multiple separate streams of instructions, thus fitting the Multiple Instructions on

Multiple Data streams (MIMD) category under Flynn's taxonomy. In order to utilize parallelism with this style of programming, a developer must explicitly specify what each stream of instructions should do. Also, there are concerns with the synchronization and creation of these threads to avoid issues with concurrency such as deadlocking and resource starvation.

In terms of hardware design, multiprocessing can be complex and difficult to scale. A CPU's processing core is complex in design, thus limiting to the number of cores that can be placed in the space of a single CPU package. For example, as of the time of this writing, a high-end desktop CPU by Intel has four physical cores that have the capability of executing as two virtual cores each [15]. These eight virtual cores allow for only eight streams of instructions to run at the same time on hardware. Multiple CPUs can be utilized in high-end computers or multiple computers can be ran together in grid computers.

A GPU is normally used for rendering images onto a computer's display. Rendering such graphics would require relatively high-speed computations of a large number of vectors very rapidly. In order to perform this computation quickly, GPUs are designed with a large number of processing cores that are very small and efficient.

In a GPGPU, a single instruction stream is executed on many concurrent threads; thus GPGPUs are classified as being Single Instruction Multiple Thread (SIMT) devices. The SIMT classification is similar to the Single Instruction Many Data stream (SIMD) classification under Flynn's taxonomy. Both SIMD and SIMT send the same instruction to many execution units, but in SIMT multithreading is combined with the SIMD model [24]. Having multithreading allows for a GPGPU to execute more tasks than in a SIMD based device.

Each individual GPU cores does run slower than a core of a traditional CPU [26]. In spite of the lower clock speed of the GPU cores, GPUs have a larger computational capability than CPUs. This computational capability is due to the relatively large number of cores that are available.

As of the time of this writing, the highest-end GPU currently available from Nvidia has 3072 cores that are capable of running computations [28]. Each one of these cores has the capability of running multiple threads themselves, which allows for the capability of a significantly higher level of parallel computing in comparison with a regular CPU. Furthermore, multiple GPUs can easily

be made to run alongside each other on a single computer, which can also increase the scale of parallelism.

1.1 GPGPU Programming

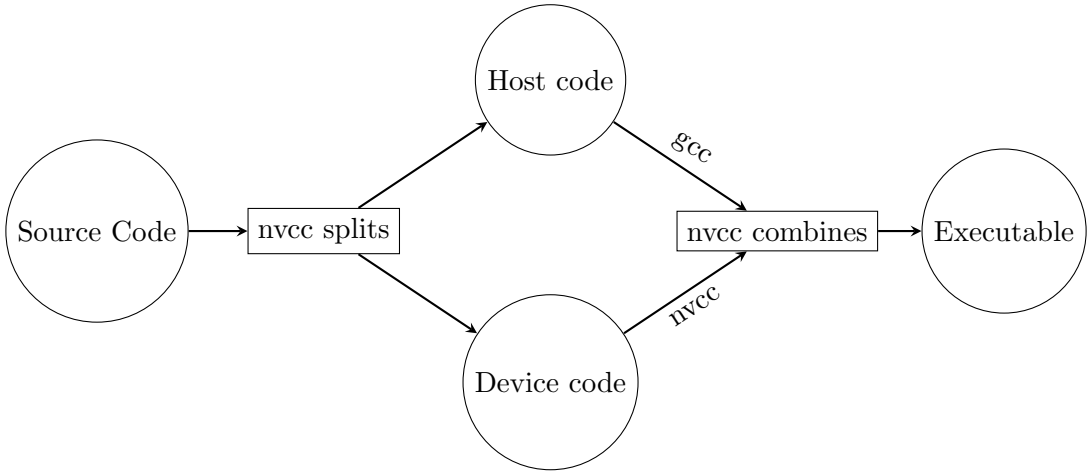

Figure 1.1. Steps to compile within nvcc.

The main existing tools for programming code that targets GPGPUs are OpenCL and CUDA. These frameworks allow for a developer to write code that will run on a CPU as well as the code that is expected to run, asynchronously, on the GPGPU. In these language frameworks, special code must be written and called explicitly to be ran on the GPU. The CPU instructions are written through a common procedural languages such as C or C++.

The GPU code is written through the *stream programming* paradigm [5]. In stream programming, the code that run on the device are called *kernels*. These kernels are functions that run on parallel GPU threads at the same time, with a different index on each thread. This is similar to writing a body of a loop, but instead of the processor running the body once each iteration with a different index, the entire loop runs at once on each GPU thread.

In the CUDA framework, these kernels are indexed through a hierarchy which can be accessed through a three-component vector. Each component identifies the threads, thread blocks, and the grid of blocks running on the system. Each grid holds a number of blocks which in turn

holds a number of threads. Currently, GPUs which are CUDA capable, are limited to having 1024 threads per block as defined by their hardware design specifications [27].

The compiler used for CUDA code is called *nvcc*. This compiler's front-end processes the CUDA source code by initially separating the code based on where the execution should occur. Next, regular code that should execute on a CPU is sent to an external C compiler, such as gcc, to process what is called the *host code*. The GPU code is compiled through nvcc to generate what is called the *device code*. Finally, nvcc combines the entire result into a single executable program. Figure 1.1 demonstrates the compilation process under nvcc.

When CUDA code is processed, the host code instructs the GPUs with the commands to run. The host code sends the device code over to the GPU through the system bus, which then copies data to be processed into the GPU memory. Next, the GPU runs the device code as a stream of instructions. When it is done, the results are copied back over the system bus to the host memory of the CPU.

In CUDA, code to be run on the GPU device are identified in the source code by the delimiters `__device__` and `__global__`. A function is callable from both the CPU host and other devices if it is delimited by `__global__`. As for the `__device__` delimiter, this function is only callable by code which is running on GPU devices; this code is not callable by a CPU host.

OpenCL is an open-source API that allows for a developer to define kernels to run on GPUs. Being open-source, OpenCL is capable of running on GPUs and parallel processing units other than those made by Nvidia. While CUDA is limited only to Nvidia GPUs, it has other performance benefits with CUDA that make it more appealing to utilize [18].

1.2 Existing Attacks

Code written for an Nvidia GPU utilizes a proprietary assembly language whose code is difficult to analyze due to the lack of accessible documentation. Also, since administrator privileges are not needed to run GPU code, powerful malware can be made to execute on GPUs in order to run without detection [30]. Moreover, since the majority of consumer computers are on x86

based architectures, most malware analysis tools have been written for such an assembly language, hindering any attempts to prevent GPU malware.

As a malware assistant, GPUs can be used to unpack malware that will run on a CPU in order to avoid detection by malware analysis tools [41]. It is also possible for the GPU to be used as a polymorphic malware extractor in order to further evasion from malware analysis tools. This concern is a challenge for a security engineer to protect against due to the lack of existing tools to enforce security policies on a GPU.

In the wild, GPUs have already been utilized for nefarious purposes. ESEA, a competitive video game service, was found guilty of hiding a bitcoin miner on their client software [32]. This miner utilized clients' computers to mine bitcoin without the clients knowledge on their GPUs while the client was waiting to start a game. The bitcoin miner introduced costly hardware damage to users since the GPUs' were made to run at higher loads than the clients would normally run their GPUs, thus overheating the devices. Other malware can utilize this idea of GPU based bitcoin mining to spread a botnet that would mine cryptocurrency for the malware writer at the price of the infected users hardware and energy consumption.

Other potential attacks can also utilize a GPU. Since GPUs have a higher processing capability than a CPU, malware can use this high processing capability. An example of a utility of a GPU for malware would be password cracking a user's encrypted files without being detected as a CPU process.

Memory on the GPU can be read allowing for an attacker to obtain information on a user's screen directly. Furthermore, this same memory can also be written to. As such it is possible to render artifacts on a user's screen to modify displayed content at will.

It was recently discovered that there was a bug in the Nvidia drivers where memory is not freed after use [2]. A user discovered that information which was rendered in his private incognito browsing instance was still available after closing his browser in the GPU memory. This user successfully wrote a program to obtain information that was rendered previously in the video buffer to verify the existence of this bug. Since the GPU did not clear memory correctly, it was

possible for an attacker to obtain sensitive information from a user's browsing history through simply reading this memories contents.

Our review of the existing works presented in this section show that there is a need for some mechanism to prevent the attacks mentioned above. Without knowledge of the assembly language, a static analysis tool of compiled code would be a significant challenge to write. As such, we propose that a language should be created to enforce security policies on CUDA source code.

1.3 Aspect-oriented Programming

This section provides a brief introduction to aspect-oriented programming. Aspect-oriented programming is a programming paradigm that will be utilized throughout this thesis. Cross-cutting concerns are the parts of a program that rely on, or are relied on, with many other components of the program. In aspect-oriented programming, code is modularized by the cross-cutting concerns. This paradigm will be utilized to simplify the insertion of security policy code into the target CUDA source code.

In aspect-oriented programming, *advice* code is inserted into specific points called *join-points*. The advice code is inserted onto the join-points through an *aspect weaver*. By having the advice code be defined separately, it is easier for a developer to write easily maintainable code. Ease of maintenance is a feature of aspect oriented programming. Since the developers can now write software without concerning themselves with the contents of the advice code while focusing on their designs. The terms and history of aspect-oriented programming are further discussed in Section 2.3.

1.4 Contributions

Through the use of aspect-oriented programming techniques we will show in this thesis that the definition and enforcement of, policies for Nvidia's CUDA platform is possible to be done. The main contributions of this thesis are:

1. The definition of simple aspect-oriented extensions to CUDA.

2. Implementation of an aspect weaver for CUDA.

3. Using the aspect-oriented extensions to weave in code segments defined policies.

4. Demonstrating that these policies are successfully enforced.

1.5 Outline of Writing

Chapter 2 presents the reader with existing related work on security policies. We present a system to enforce security policies on GPUs in Chapter 3. Examples of using the system created in this thesis are provided in Chapter 4 along with a performance analysis of the effects introduced by utilizing this system. In Chapter 5, we provide goals for future versions of the system which we presented. Lastly, in Chapter 6, we present future work and summarize the thesis.

CHAPTER 2

RELATED WORK

An overview of the existing works relating to security policy specification will be presented in this chapter. Firstly, the scope of theoretical work involved with security policy studies will be introduced. A survey of existing languages and tools for policy specification will be presented where we shall show that there does not exist a tool suitable for the needs of GPGPU security. Lastly, we will present in detail a useful programming paradigm that we have utilized for the system developed in this thesis.

2.1 Security Policy Theory

Security policies are executions that are determined to be unacceptable by some pre-defined rules. Enforceability theory is the study of what can and cannot be enforced as a policy through policy enforcement mechanisms. These policies might relate to general purpose concepts such as *access control*, *information flow*, and *availability*. System specific and special purpose policies are very important utilities that would be used by a security engineer in order to define how a system should behave.

An example of an application specific security policy is the traditional Unix file system permission. These file system permissions enforce access control by checking the file system flags and determining whether or not a user has the authority to read, write, or execute a file [31]. The operating system that is enforcing this policy prevents any requests to files by users without the correct permissions, thus preventing any unauthorized file accesses.

There are two classes of properties, these policies are known as *safety* and *liveness* properties [21]. Policies that specify that "nothing bad happens" are defined as safety properties. On the

other hand, a liveness property is a policy which states that something "must" happen in order to satisfy the policy's requirements.

Schneider formalized program monitors and discovered limitations on them. Furthermore, he presented a class of enforcement mechanisms that he defined as *Execution Monitors* [33]. This class of monitors can only enforce *safety* properties. This class of enforcement mechanisms works by monitoring the execution steps of a system which he defined to be the *target*. When the policy was violated, the Execution Monitor would terminate this target system. In his work, he also defined the class of security policies that are enforceable by Execution Monitors. Specifically, it was established that mechanisms which use information that would be unavailable from monitoring an executions trace are to be excluded from Execution Monitors.

From the definition provided by Schneider, Execution Monitors would not be capable of enforcing a policy that requires knowledge of an execution's future steps. As such, Schneider excludes compilers, type checkers, and other forms of code static analysis from the Execution Monitor family of enforcement mechanisms. This is because having knowledge of alternate paths of code and future execution is the feature that invalidates the definition of these tools from being Execution Monitor mechanisms.

For non-safety policies, Ligatti built on Scheider's definition of policies [23]. Ligatti defined the theoretical monitors as being modeled as *edit automata*. An edit automata is a *transformer* of code which has the capability to insert, or suppress, actions on behalf of a target. Having this capability allows for a remedial action to be specified for an execution instead of simply rendering a trace as invalid. These remedial actions allow for liveness policies to also be enforced.

2.2 Policy Specification Languages

A number of policy definition tools already exist to specify and enforce security policies. At the time of this writing, there was no security policy specification language in existence that we have found that can be used for GPGPU computing. This section will introduce the scope of research of policy specification languages and summarize selected languages.

Polymer is a specification language and system for composable security policies in Java. Polymer's implementation deals with complex security policies by allowing a security engineer to specify a security policy as the composition of smaller sub-policy modules for enforcing policies. Polymer has three core abstractions: actions, suggestions, and policies [4]. The *action* objects include all the information that is relevant to security sensitive method invocations. *Suggestions* are used to suggest ways for the monitor to handle actions which trigger a policy constraint. The *policies* are the monitors themselves that query actions, can accept suggestions, and then return a result based on the suggestion.

Ponder2 is an object management system that allows for inter-object message passing in a distributed system [40]. Based on an existing language, Ponder, the system Ponder2 aims to incorporate events and policies to object management [9]. Policies in Ponder2 are defined to the basic types being either *obligation policies* or *authorization policies*. Obligation policies are actions that *must* be performed by the systems managers when an event occurs based on a set of conditions. Authorization policies are simply rules to allow, or deny, message passing between objects. An authorization policy is essentially an access control policy. In Ponder2, the policy can either be designed to allow or deny.

Members of a system in Ponder2 are organized into *domains*. These domains simply act as containers for objects in a hierarchy. Policies in Ponder2 are specified in terms of domains instead of directly on the objects within a domain. When a policy is specified onto a domain, the same policy is inherited to the sub-domains. However, when a domain has multiple parents, the domain takes the most explicit policy and does not combine these policies.

The eXtensible Access Control Markup Language, XACML, is used for specifying access control policies [14]. In XACML, a policy is defined with three components: a *Target*, a *Rule set*, and a *Rule combining algorithm*. A XACML Target is defined as a set of requests to which the policy applies. The Target can be inherited from the Parent policy if a XACML rule is formed through combination. The *Targets* in XACML are defined statically. The rules in the rule set each have another (optional) Target, Conditions, and Effects. The Condition defines what restrictions are needed for the Effect to occur. XACML is limited as a policy language to simply permit or deny

in access control as effects. It is also possible for a policy in XACML to include *Obligations* which are functions to be executed that may affect the decision. Finally, the Rule combining algorithm defines how to resolve conflicts when a policy is combined with another policy.

Run-time enforcement in a distributed system has used to enforce security policies. Other research involved in the area of distributed security policies can be found in [39, 20, 38, 35, 34, 17, 10]. These works have defined how to communicate enforce security policies in a distributed environment. However, none of these works have defined policies, or policy enforcement mechanisms, for GPGPUs.

2.3 Aspect-Oriented Programming

In order to allow for a developer to write the CUDA based code without having to concern themselves with the security, we have decided to take an aspect-oriented programming approach. This section will serve as an introduction to terms and concepts needed to explain aspect-oriented programming, provide the background work for aspect-oriented programming, as well as attempts to apply aspect-oriented programming to GPGPUs.

Aspect-oriented programming is a programming paradigm where a program is modularized by its cross-cutting concerns. The concept of aspect-oriented programming was explicitly defined by Kiczales in [19]. An example of aspect-oriented programming is an extension to the Java language known as AspectJ [13]. In order to discuss aspect-oriented programming, related concepts will be defined in his section.

When two properties that are being programmed have to be built differently, yet coordinated in some way, there can be some difficulties with the interfacing of these properties. We say that two properties *cross-cut* each other when they have to be coordinated in such ways.

If a property of a program can be fully expressed in a general procedure, it is defined as a *component*. Components can be easily composed as needed and are easily accessible. These components are considered to be individual units of a systems' functionality.

In general purpose language paradigms, such as procedural or object oriented, a programmer cannot build systems which cross-cut each other independently. This leads to difficulty for a

programmer to compose two properties together without having the code tangled in difficulty to decompose ways. The difficulty is because if the specifications change in any of the properties that are the results of a composition, the programmer has manually decompose the components, implement the modifications needed, finally recomposing the code.

Advice is any additional behavior that should be applied to the existing program. Advice can be any behavior such as, but not limited to, additional code, performance or access patterns, or even logging functions. In this paper, all of the policies to be implemented are considered to be advice since they are going to be behavior added to an existing CUDA program.

A *join-point* is a point in a programs execution path where the code, advice, from an aspect should be performed. The set of join-points is known as the *point-cuts* in aspect-oriented programming.

The combination of join-points and advice are *aspects*. In order to place aspects in their correct locations in code, an *aspect weaver* is used. An aspect weaver takes the set of aspect and component programs outputs another program as output. This program would have all the advice at the correct join-points in the flow graph.

In terms of the CUDA language framework, there has not been any existing implementations, as far as we are aware, where aspect-oriented programming was actually applied to CUDA. The utility of aspect-oriented programming has been proposed as a desirable future feature for CUDA to allow for performance benefits [29]. Also, aspect-oriented GPU programming has been considered for simplifying programming through languages that output CUDA from object-oriented languages such as in [12, 42, 25]. However, these existing works do not actually implement aspect-oriented programming to existing CUDA code, but instead they take existing languages, which are not CUDA, and use aspect-oriented programming techniques to output code that can be compiled under a CUDA compiler.

There already exists an aspect-oriented programming implementation for the OpenCL framework [8]. However, as far as my research has found, there is no available implementation for CUDA code. Even though the OpenCL version can run on more hardware, we have chosen to

continue the route of using CUDA due to the added performance of running CUDA code and due to my familiarity with CUDA compared to OpenCL.

2.4 Chapter Summary

The related research has been described in this chapter in order to allow the reader to familiarize themselves with the relevant terms and understand the current state of works of security policy enforcement. Enforceability theory as a field has been deeply studied in the literature.

Many tools and languages exist for specifying security policies. From the set of policy specification languages, none of the existing ones found have considered GPGPUs as a target platform on which to enforce security policies.

Also, aspect-oriented programming as a paradigm was introduced with the terms defined thoroughly. It was shown that aspect-oriented programming currently does not exist for CUDA. Furthermore, it was demonstrated that there exists a desire for aspect-oriented programming for CUDA for purposes outside of the needs of security policy specification.

CHAPTER 3

PROPOSED SYSTEM

This chapter we will provide a detailed description of the main contributions proposed in this thesis. The CUDA aspect weaver that we introduce will be introduced in this chapter. A list of possible policy types for this system is defined. Also, the process of writing policies for this CUDA aspect weaver will be explained. Finally, the development process of using this weaver with existing CUDA code will be discussed.

3.1 The eGASP Policy Enforcement System

In order to enforce defined policies on a target CUDA program, it was decided to weave in the policy code onto the source code of the target program and output a CUDA program which has been enforced. This approach allows a CUDA developer to work independently of a policy engineer as the policy code is kept separated from the target CUDA code. As was discussed in Section 2.2, weaving code is a standard practice used for policy enforcement, e.g. the policy enforcement mechanism in Polymer [4].

The name `eGASP` was chosen for this policy enforcer. This name is an acronym which stands for "enforcing GPU policies through Aspect Style Programming". The `eGASP` program will take a `gasp` file and weave in the policies into the target code at specified join-points, locations in the target code to insert aspect code, to enforce the specified policies.

This policy enforcer program was written using Python. Python was chosen to utilize existing string manipulation libraries to inline the policies into a target CUDA device function. We provide the full source code of `eGASP` online for use [1].

Table 3.1: GASP policy types.

Policy type	Join-point location.
Pre-execution	Into the function at the beginning.
Pre-return	Prior to all instances of `return`.
Post-execution	At the very end of the function.

```
@begin signature
@preExec
    Pre-execution code
@preReturn
    Pre-return code
@postExec
    Post-execution code
@end
```

Figure 3.1. File format specification for a gasp policy.

There are three types of policy join-points supported in `eGASP`. For eGASP the three policies are *pre-execution*, *pre-return*, and *post-execution*. The target code is the file onto which the policy needs to be enforced, and join-points are where the policy code needs to be inserted. These policy types define the join-points in which code will be inserted verbatim. A description for the types of policies is found in Table 3.1 where the join-points are defined for each policy type.

A policy engineer can write a policy for `eGASP` in a file with the file type extension `gasp`. In a `gasp` file, the engineer has to define which function to enforce by providing a function *signature*, and the aspect code to be inserted at the join-points specified. The format of a `gasp` is defined in Figure 3.1. In this format, all of the sections need to be included for every policy type in a `gasp` file.

If a policy engineer does not need to enforce any policy for a specific policy type, they may leave that policy section blank. The policy engineer still needs to define all sections of a `gasp` file. Furthermore, a policy engineer can define policies for multiple target functions in the same `gasp` file by writing a full `gasp` specification for every policy with each section defining all parts of a policy.

The signature of a target function has to be, verbatim, what is written in the target CUDA code to declare the function. This includes the `__global__` or `__device__` identifiers which are required to define a function as mentioned in Section 1.1. The `eGASP` program would search for

the function that matches this signature and then proceed to insert the code at the join-points specified.

```
// This placeholder is empty
// __global__ void sensitiveFunction(int *input)
// {
// }

__global__ void sensitiveFunction(int *input)
{
    // This function's body should not execute.
    maliciousFunction(input);
}
```

Figure 3.2. Function in a comment.

A distinction has been intentionally made between pre-return and post-execution code to fit a specific edge case of defining security policies for functions with a `void` return type. Any `void` function can use a `return` statement to break out of execution early. Having such a distinction allows a policy writer to specify different policy behaviors based on whether a target function returns explicitly or exits from the end of execution.

Prior to any code insertion, `eGASP` strips away all of the comments in the target code. Comment stripping is a standard behavior for the GNU C compiler in the preprocessing phase as defined in the documentation [36], thus removing the comments would not affect the functionality of the code. The reason for stripping away comments from the target program in `eGASP` is to prevent a possible malicious programmer from avoiding the policy enforcement altogether. If comments are not stripped, this form of policy enforcement circumvention would have been possible by placing a fake function with the same signature in a comment located prior to where the actual target function body is written in the CUDA program.

An example of a scenario where a malicious developer might attempt to avoid a policy being enforced by `eGASP` through hiding in a comment is presented with the code block in Figure 3.2. In this example, a malicious code writer knows that the security engineer has a policy to exit out of a function prior to executing its body. The malicious user could have attempted to put an empty placeholder function with the same signature as `sensitiveFunction` within comments. In

this function body, another malicious function could be called by the programmer, resulting in violation of the security policy.

If comments weren't stripped away from the code, `eGASP` could only modify the first occurrence of `sensitiveFunction`. This would allow the malicious code writer to execute their malicious functionality even after a policy engineer would have attempted to have this code be enforced. Since `eGASP` removes all comments, the code that is commented will be removed from the source. Stripping the comments renders such a malicious avoidance irrelevant as `eGASP` would enforce the correct code and not be affected by the commented code.

3.2 Utilizing eGASP in a Development Environment

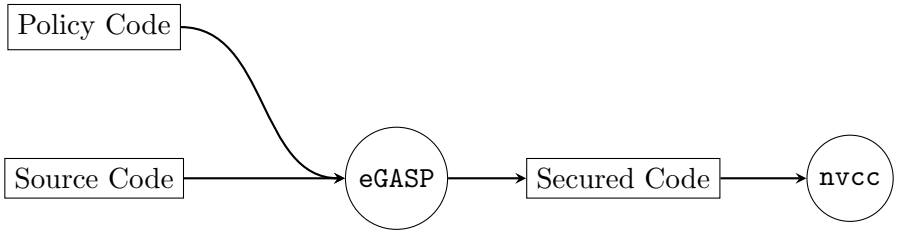

Figure 3.3. Development workflow with eGASP.

For a CUDA developer to use `eGASP` in their workflow, the developer would require access to both the source code file of the target program, and the `gasp` file with the policies. The process of using `eGASP` in a development workflow is illustrated in Figure 3.3. The source code and `gasp` files would be inputs to `eGASP`. Running `eGASP` would output a CUDA source program. This resulting program can be compiled through `nvcc` instead of the original target program to obtain executable code which has been fully enforced by the specifications defined in the `gasp` file.

A systems engineer in an organization can automate this process by replacing `nvcc` with an alias to a script that performs all the steps in Figure 3.3 with a system-wide policy that is given by the policy engineer. By creating such a script, using `eGASP` would be transparent to the developers in the environment, and we believe this would not interfere with the process of CUDA development.

3.3 Chapter Summary

This chapter introduced the GPU policy enforcement system `eGASP` with descriptions of how to write policies and use `eGASP`. The possible policy join-points which can be enforced are defined for the aspect weaver as being pre-execution, pre-return, and post-execution with reasoning for distinguishing pre-return and post-execution being given. A file format specification for writing policy code in a `gasp` file for `eGASP` is defined for policy engineers. There was a possible method to avoid `eGASP` which was discovered during development, but the solution to that vulnerability was described with the mitigation implemented. Lastly, the compilation steps of `eGASP` are described showing what a developer would have to do to use this system.

CHAPTER 4

EMPIRICAL EVALUATION

Examples of `eGASP` policies and the results of testing these policies are presented in this chapter. First, the hardware configuration that we used to perform these tests will be presented; this configuration can be used to reproduce tests. Then, some policies to emulate what a security engineer might wish to enforce in various situations are evaluated. Finally, a performance analysis of using `eGASP` to enforce policies compared to a manual method of code insertion will be measured.

4.1 System Configuration

Table 4.1: System configuration.

Component	Specification
Operating System	Arch Linux
Kernel Version	4.4.5-1-ARCH
Desktop Environment	Gnome 3.18
Driver Version	361.28
CUDA release	7.5, V7.5.17
CPU	Intel 2600K overclocked to 3.5 GHz base, 4.43 GHz max
GPU	Nvidia GTX-580
RAM	16-GiB DDR3

All of the tests in this chapter were run on a personal desktop with the specifications detailed on Table 4.1. A reader can reproduce the results of all of the tests presented in this chapter given the specifications provided. This desktop was chosen due to availability of hardware at the time of writing this thesis.

4.2 Examples of Policies

Several policies have been written using `eGASP`. These policies will be listed in this section with the utility explained. The last policy will be used to demonstrate the performance difference between manually inserting the policy code compared to using `eGASP`. As was mentioned in Section 3.1, the policy engineer needs to include all sections in a `gasp` file, thus all of the figures in this chapter will include an entire `gasp` policy.

4.2.1 Logging

```
@begin __global__ void Sensitive_Global
@preExec
    long long int startTime = clock64();

@preReturn
@postExec
    timeLog[blockIdx.x] = clock64() - startTime;

@end
```

Figure 4.1. Policy to log execution time.

Logging in a common policy desired for security and policy engineers. To emulate the functionality of logging, we wrote a policy to time the execution of the body of a function. This policy will save the calculated execution time in an array that was passed in as a parameter. When the policy returns, the calling code can then check the contents of the array `timeLog` for the results. This `gasp` policy is shown in Figure 4.1.

To facilitate logging, the program had to be manually modified to pass in an array `timeLog` to the function called `Sensitive_Global`. This requires manual intervention of the developer to allow for the policy to be enforced onto a target. A proposed method for enabling this modification without manual intervention of the developer is explained in Section 5.3.

```
@begin __global__ void Sensitive_Global
@preExec
    if (blockIdx.x > 300)
    {

@preReturn
@postExec
    }

@end
```

Figure 4.2. Policy to limit blocks which execute.

4.2.2 Imposing Execution Limitations

It was decided to demonstrate how to define and enforce a safety policy through the use of `eGASP`. Safety policies are policies which specify that nothing bad happens. In this policy, we prevent the execution of a CUDA function from continuing after it violates our rules. Such a policy would be considered to be a safety policy according to this definition. The theory behind safety policies was described in Section 2.1.

The purpose of the proposed policy was to limit the number of CUDA blocks on which the device code would execute. This policy can be used as a form of load limiting of code where each block can run fully independently of each other. The method of doing this execution limiting is performed by wrapping the body of the target function with a check of the index of the blocks. If the check fails, then the body of the function would not execute. The execution limiting policy is shown in Figure 4.2.

It must be noted that in order to wrap the body of the target function, a brace must be opened in the `@preExec` point-cut location, and closed in the `@postExec`. The braces have to be placed in such sections since `eGASP` inserts the advice code from the policy verbatim into the locations specified. These braces wrap the existing body expression of the target function as the body of an `if` conditional statement, as such the body would be jumped over if the condition could not evaluate correctly.

```
@begin __global__ void Sensitive_Global
@preExec
        int thisID = blockIdx.x + blockIdx.y;

@preReturn
@postExec
@end
```

Figure 4.3. Policy to complete missing code.

4.2.3 Completing Missing Code

We designed the policy in Figure 4.3 to demonstrate that `eGASP` can be used outside the realm of security policies. For educational purposes, a class teaching CUDA to beginner programming students might wish to abstract away concepts early on from the students to ease their learning. An example would be to only show the relative functional body of code to these students while eliminating the concern of calculating the indexes of each thread for a later lesson.

To build a policy that that represents this classroom situation, we moved the process of calculating the index of a CUDA block into a policy in a `gasp` file. Having the calculation in a separate file allows us to abstract that portion away from the body of the CUDA code as shown in the `gasp` file defined in Figure 4.3. This will allow the students to write code for CUDA that will compile without having to worry about calculating the thread indices. The calculated index in this policy is stored as an integer to be referenced by students as the variable `thisID` throughout their code.

The `gasp` policy defined in Figure 4.3 can also be used for CUDA developers in a large organization. An algorithm designer in this organization might wish to write an algorithm for CUDA without worrying about the geometry of the kernel being called. By having the index calculation in a separate `gasp` file, this designer can define the indices after fully designing their algorithm without modifying any of their CUDA code. Furthermore the algorithm designer can change the geometry of their algorithm when the organization purchases different hardware without having to go into the source CUDA code.

4.3 Performance Analysis

```
@begin __global__ void simple_function
@preExec

    // local array to represent generated primes
    int localArray[10001];

    // Initialize entire array as "true"
    for(int i = 0; i < 10001; i++)
        localArray[i] = 1;

    // Starting at 2 untill sqrt(arraysize)
    for (int i = 2; i <= 100; i++)
        if (localArray[i] == 1)
            for (int j = i*i; j < 10001; j+= i)
                // These numbers are not prime
                localArray[j] = 0;

@preReturn
@postExec
@end
```

Figure 4.4. Performance testing code.

In this section, we will present the results of some performance tests that were performed. To artificially introduce measurably long execution time, each CUDA block calculates the first prime numbers which are less than 10,000 by using the Sieve of Eratosthenes. Moreover, manual tests were performed to compare inlining the aspects of a policy against inserting function calls to inserted functions containing policy code. The results of the performance tests described in this section are summarized in Table 4.2.

Figure 4.4 provides the definition of the policy which was used to insert a block of code into a test CUDA function called `simple_function`. This policy was specifically designed to have a slow execution time when run in order to allow us to measure if there is a significant performance difference between using `eGASP` compared to manually inserting the code. To facilitate logging, we utilized the `cudaEventElapsedTime` function that provides us with a resolution of 0.5 microseconds [27].

Table 4.2: Measurement results.

Process	Execution Time
Inlining a function	483.018 ms
Calling Function	485.305 ms
Calculating Sieve of Eratosthenes	480.408 ms
`simple_function`	41.938 ms
Enforced Target	482.746 ms

At its current state of design, the `eGASP` weaver inlines the policy code onto the target program. This choice to inline code was made to avoid any potential overhead of function calls being detrimental to a target CUDA functions execution time. We have expected that the execution time of inlining the aspect code onto a target program would not be significantly higher than that of a target program calling the aspects in separate functions.

When we tested the runtime of inlining the policy code in Figure 4.4 to a target program against having the policy code be placed in a separate function that gets called, the average time to call a body of a function was 485.305 ms after 100 iterations. As for inlining, it took an average of 483.018 ms to execute 100 iterations which is approximately 2 ms faster than using function calls. However, with these results, we consider the difference of execution time between inlining the policy code onto a target against calling a device function from the CUDA target code to be insignificant, as it is less than half a percent difference in execution time.

It was expected that there would be some level of overhead to call a CUDA device function from the CPU host. This overhead is due to how, in the CUDA framework, the CPU host needs to transfer instructions to the GPU over a relatively slow PCI bus [27]. As such, we expected that it would be faster to execute run code in a policy instead of having to return to a CPU to perform operations between calls to the GPU.

In our tests of this claim we also have timed how long it would take for 100 iterations of `simple_function` to be called from host code and complete executing. We have found that average execution time of `simple_function` was 41.938 ms. The runtime of a function that only has the contents of the policy in Figure 4.4 was found to be 480.408 ms. When we applied the policy of generating primes upon `simple_function`, we have found that the enforced target was less

than the sum of execution times of a function to perform the generations and the time to execute `simple_function`, therefore verifying our claims.

4.4 Chapter Summary

This section provided examples of policies for a developer who intends to use `eGASP`. We demonstrated policies that can be used to enforce logging, a safety policy that limits execution based on a defined rule, and a policy that can be utilized for simplifying the process of writing CUDA code.

We also tested the performance overhead of the code that has been enforced through the use of `eGASP`. Comparing target code that has been enforced `eGASP` to manually inserting the policy code and found no significant performance penalty when using `eGASP`. Finally, we have demonstrated that the design choice of inlining the policy code of a `gasp` file onto a CUDA program, would not have a significant detrimental effect on the performance of the target program compared to using function calls for the policies.

CHAPTER 5

IMPROVING THE POLICY ENFORCER

At its current state of design, the `eGASP` weaver is a proof-of-concept tool. Developing our policy enforcer for CUDA has introduced difficulties since there does not appear to be an open-source compiler available at the time of this writing [6]. To make `eGASP` complete either a full-fledged CUDA compiler needs to be written, or the functionality of `eGASP` needs to be inserted into an existing compiler. This chapter lists some features that can be considered to be desirable for a policy engineer to utilize. Each of the following sub sections details a proposed feature for `eGASP` and the techniques which can be used to apply such features.

5.1 Join-points Prior to Calling CUDA Functions

A potential policy type that a security engineer may wish to include when designing their policies might require processing data prior to calling functions. This could involve running a predetermined sequence of code prior to all of the CUDA function calls that the engineer specifies to be targets. It is possible to find all possible entry points to a CUDA device function in a CUDA program through an exhaustive parsing of the target programs source code.

If all the correct function calls are successfully located by traversing the tree resulting from the parser, it may be possible to place advice code prior to a target functions calls as join-points. The intended aspects in the policy code can be weaved in later into the target code through a trivial inline insertion of the aspect code at the entry points located from the parser.

One proposed situation where this form of policy is more desirable than the pre-execution policy described in Section 3.1 is in a high performance and time sensitive computer system environment. There is a level of delay every time that a CUDA kernel is invoked from a CPU device [7].

By having a conditional prior to entering the body of a CUDA function instead of within the body of the CUDA function, we can avoid this delay. It may even be argued that a developer can utilize a policy as a performance optimization can be designed through utilizing this technique.

5.2 Join-points After Returning from CUDA Functions

As in Section 5.1, the locations of the entry points to a CUDA function can be found through parsing of the source code. Given the entry points, we can use the same process of parsing the source code of a program to locate points from a CUDA program where target functions return.

Having join-points after returning differ from the feature in Section 5.1. The contrast would be that, in this style of a policy, the aspect code insertion should be executed after the function call is complete, to ensure that the advice will run after returning from the calling function. This type of policy is not the same form as the `pre-return` code defined in `eGASP` in Section 3.1 as the post-return style of policies target execution location exist outside of the functions body.

Post-return code can be useful for logging and post processing of data after a GPU has given results. An example of a desirable functionality from a policy where logging can be performed is using information given after running a CUDA kernel and use the CPU to process this data to save to a file. This logging to a file policy cannot be written at the current state of `eGASP` as `eGASP` only manipulates CUDA device code. This is because device code does not have access to file operations [27].

5.3 Modifying the Arguments of a CUDA Function Signature

In the logging policy, described in Section 4.2.1, manual preprocessing was needed before enforcing. If a policy can pass in more information to a CUDA function, a policy engineer can easily implement logging code without manual intervention. To enforce that policy, manual intervention was required by the developer to add a parameter to the function in various points. By empowering the system engineer to add more arguments to a CUDA function, the system engineer can build more diverse policy rules since the policy engineer can add an argument just for the policy for the means of information passing.

Conditional code execution is an example of a form of a policy where information passing through arguments can be made possible. With this information being passed, a policy engineer can have the policies determine behaviors based on previous states, where the state information is saved onto the variables. These same variables can be passed to other policies to define a behavior based on previous states.

As an example where information passing can be used in a policy, a boolean flag variable can be passed to ensure that a code segment only executes once. This flag would be set upon the first execution of a function to be signify that a segment has already executed. A policy within the body of the function can now check the state of that flag to determine whether or not the function should execute a section.

A parser can be used to find all the locations where the target function is called. Having obtained the function call locations, we can manipulate the arguments for this function along with the parameters in the function declaration and prototypes throughout the code. However, care must be taken with this technique to avoid causing a conflict with other overloaded function definitions.

There exists a limitation with locating function pointers statically. Static analysis of pointers to identify their runtime values is an undecidable problem [22]. Manual identification would be required by the user to identify where function pointers would point to at runtime for `eGASP` to enforce code with function pointers.

It might be possible that a developer has defined multiple functions with the same name that are overloaded by the argument count of the function. If the added arguments to the target function end up causing two functions to have the same function signature there would be a conflict between the two functions, thus the resulting program would not compile.

One possible solution for this issue is to add empty arguments to the end of the definition of the target function to ensure that there would not be a resulting conflict. These arguments would exist as placeholders that have no effect on the execution of the program. If adding arguments has created a function definition that matches an existing function, we can add more empty variables until we ensure that the resulting function does not match any pre-defined functions.

5.4 Handling Classes and Types

Currently, `eGASP` performs a blind search and replace for functions based on their signature. It is possible that two separate classes can have functions with the exact same signature, but are distinct themselves. Another concern arises when classes inherit from each other and they overload the function definitions. A policy engineer would not have any way to differentiate between these distinct functions.

It could be beneficial for a policy engineer to specify a policy where a function would have different policy behaviors based on the function's run-time type. For example, the policy engineer could know that a program dictates different behaviors of a function depending on which type it is. The policy engineer could then be more explicit in defining rules for the target program for each type. Preserving type safety in the behavior of the program is one reason that a policy engineer might wish to have a different set of rules per run-time type of a function.

We can utilize a join-point model inspired by AspectJ to implement the capability to handle classes and types of a policy for `eGASP`. AspectJ is an aspect-oriented programming extension for Java that has a join-point model where the run-time types of objects are checked to determine the join-points [13]. A type checker would be needed to perform these forms of policies in `eGASP` to check for the type of a CUDA function. Furthermore, dynamic type checking would also be needed for programs that utilize polymorphic functions as the type of a function might change.

5.5 Composing Policies

The composition of a security policy through *polcy combinators*, combining algorithms, enables a security engineer to design very complex policies through the combination of multiple, simpler to write, sub-policies. Polymer is an example of an existing policy enforcement mechanism that allows for security engineer to design the complex policies in such a way. The Polymer policy enforcement and XACML are frameworks that allow for policy code reuse through policy combinators [4].

Policies in eGASP need to be defined on a per-function basis. A security engineer might wish to enforce a policy on multiple target functions running the same behavior, or the capability to have multiple, already defined, behaviors be used to create a new and more complex policy. The process of enforcing multiple targets with the same behavior with eGASP currently would require for the engineer to manually create a policy with the exact same content for every function.

Conversely, if an engineer wishes to create a new policy that is based on a composition from an existing set of policies, the policy engineer needs to manually define the composed behavior as a brand new policy. Neither of these processes allow for the ease of defining complex policies as there is copious amount of manual code writing overhead to define the policies in either task. Having these difficulties of writing large policies would make an engineer's task of defining policies for a large system be an acutely tedious engagement to deal with.

We believe that if eGASP was enhanced to allow for the capability of policy composition, a policy engineer's duty would be simplified. Composition would lend a policy engineer utilizing eGASP the ability to pre-define small sub-policies that would encourage program modularity similar to libraries in common languages. Furthermore, a team of policy engineers could distribute the workload of policy definition to having a dedicated engineer for every sub-policy which will be composed later, increasing the efficiency of policy definition.

5.6 Future Plan for Implementing Additional Features

The features which were listed in this chapter would benefit a security engineer but would be difficult to implement without having access to a compiler's internals. This section will serve as a proposal for the methodology that may be used to implement the features which were mentioned in the previous sections.

Many added features for eGASP have been shown to be desirable. Access to a CUDA compiler's internals would ease the development of eGASP to incorporate the features proposed in this chapter. Being able to leverage the capabilities of the existing parser from a compiler would be beneficial to implement the features in Sections 5.1, 5.2, and 5.3. Having access to the parse tree would ease the process of locating entry and return points of a CUDA function, as well as allow

eGASP to insert aspects as policy code onto these join-points of the program. Furthermore, having such access would allow for eGASP to identify overloaded functions, thus avoiding the potential issues which were mentioned in Section 5.3.

Access to the type checker can be utilized in for the concerns described in Section 5.4 to identify the exact type of a function. A compiler performs static type checking to ensure that the program would not have violations in the type system. By leveraging this type checker, we can equip eGASP with knowledge of a target CUDA functions type to ensure that that the policies to be enforced are in fact the types that we expect them to be. Also, we can utilize the compiler to insert run-time checks for a functions type to allow for a policy to enforce a polymorphic function with a behavior that is dependent on the function's run-type.

Rules for composition need to be formally defined for eGASP. These rules need to take into consideration the parallel nature of software running on the CUDA framework to ensure that undefined behavior is not an accidental by-product of a policy. Finally, the composition rules need to be clear and concise as to ensure that the resulting policy of a composition is what would be expected by the policy engineer after writing a complex security policy.

5.7 Chapter Summary

Several desirable features were discussed in this chapter that we were unable to implement in eGASP. These features include: join-points prior to calling a function, join-points after returning from a function, the capability to modify a CUDA functions signature, handling polymorphic functions in CUDA, and the ability to write policies through the processes of composing smaller sub-policies. Enforcing code with function pointers would require that a user manually identify what the runtime values would be.

A plan for eGASP to be extended with all of the features mentioned in this chapter was given. In that plan, it was argued that the implementation of the desirable features can leverage the inner workings of a CUDA compiler. To utilize the inner workings, it is also vital that the source code of the compiler be freely available for the ease of implementing the proposed features for eGASP.

CHAPTER 6

FUTURE WORK AND SUMMARY

In this chapter we will present what we believe are the next steps to progress with research on GPGPU security policies. Opportunities for research on CUDA policy design are listed as future work. We will be presenting a competing platform to CUDA and explain why we believe we can apply `eGASP` to that platform. Lastly, we will provide a summary of the writing presented thesis.

6.1 Future Work

We have demonstrated a few policies utilizing `eGASP` to enforce CUDA programs in Chapter 4. More policies need to be created and tested for CUDA using `eGASP`. Further testing to identify what styles of policies a security engineer would be beneficial to identify the realm of policies which can be applied to a GPGPU.

There were some more policy types that we wished to implement in `eGASP` aside from the ones listed in Section 3.1. We have listed out the desired policy types in Chapter 5. It was also argued that it would be possible to implement the features if we had an open source compiler in a detailed plan that we have laid out in Section 5.6.

Implementing the open-source compiler would also provide another security improvement. As the last step of building a program with `eGASP` involves utilizing the nvcc compiler, ultimately we are left to trust nvcc to handle the output code. Without a means to verify the compiler, we are not capable to fully trust the output [37]. Having the source of the compiler would allow us to verify, within a reasonable certainty, that the compiler in itself is not malicious.

During the design of `eGASP`, we have considered making `eGASP` work as a parallel enabled Execution Monitor running alongside of other processes. As was described in Section 2.1, Execution

Monitors are a class of enforcement mechanisms that monitor the execution steps of a target and terminate the target upon policy violation. We wished for this Execution Monitor to monitor multiple tasks at the same time. Unfortunately, as we describe in the following paragraph, we cannot have an Execution Monitor.

In CUDA, threads on a grid run concurrently to each other. These threads cannot execute code where the execution path could diverge [27]. As such, we have argued that if a single thread violates a policy while the remaining threads do not, we cannot target only singular thread for a remedial action. Also, due to high degree of parallelism of GPGPUs, it would be difficult to synchronize execution an Execution Monitor without incurring massive amount is introduced slowdown.

We have looked at other parallel computation modules for this concept. Intel provides its own family of highly parallel computation platform as a competitor to CUDA and OpenCL. The Intel Xeon Phi family of coprocessors allows for highly parallel computation through traditional programming languages and techniques [16]. With the Phi coprocessor, a programmer has a higher degree of flexibility of design of their programs. While in CUDA we are limited to *stream processing*, with the Intel Phi we are allowed a larger amount of flexibility to decide on programming models.

In the Phi coprocessor there are far fewer cores than in a GPGPU. Compared to the 3072 cores in the current top-end consumer GPU, the top end Intel Phi has 61 cores [16]. These cores also have fewer threads per core than in a GPU. Overall the total thread count would be much lower on a Phi than on a GPU, thus reducing the degree of parallelism.

Nevertheless, each of the cores on a Phi are based on standard Intel processors in the underlying design, inheriting several desired attributes. The processing cores of a Phi can execute the same instruction set as a traditional x86 processors. This permits a programmer the capability to write a program which runs on a CPU and is binary compatible with a Phi coprocessor.

Phi allows for parallelism through the SIMD style of computing architecture to enable for stream processing capabilities. A programmer can write software targeting the Phi using stream processing using OpenCL as well. Programming in OpenCL allows for a developer to have the

capability of writing the same of programs for a GPGPU to run on a Phi, albeit with a much lower level of parallelism.

An Intel Phi does have the benefit of being more flexible than GPGPUs. It is possible for a developer to treat each core of a Phi as its own host in a cluster. This is due to how in a Phi, each core can run a micro Linux operating system in an internal, high speed, network. As such, the Phi opens opportunities to utilize standard methods of parallel computing on a single device. An example is the message passing interface which allows for parallel computation through a distributed computing platform [3].

Through utilizing an Intel Phi coprocessor we propose that we can build a more capable policy enforcement mechanism than `eGASP` at its current state. The enforcement mechanism can run in parallel to the target code running on the same coprocessor. This proposed enforcement mechanism can be designed to be a dedicated execution monitor running as its own thread on the Phi.

We believe that it can be possible for this parallel execution monitor to performing checks on multiple, independent, processes running on the Phi. Furthermore, since each core of the Phi operates with its own micro operating system, the cores have native support for file operations. Native file operations will enable the policy designer with the capability of having a distributed logging policy.

Finally, we know that the Phi can be programmed using standard C [16]. Also, we can compile programs designed to target the Phi using the GCC compiler. Since we have the full source code of GCC to work with, applying all of the features of `eGASP` as well as the features listed in Chapter 5 is would be possible for the Phi.

6.2 Summary

General Purpose Graphics Processing Units are high performance processing devices. We have identified existing and theoretical security attacks involving GPGPUS. Some of those attacks have even utilized a GPGPU as an aid to hide their malicious executions.

The field of aspect-oriented programming as a paradigm to increase software modularity was presented. Aspect-oriented programming techniques were used to create our security policy enforcement mechanism called `eGASP`. We have demonstrated some policies to not only enforce security policies, but also to ease the development process of an engineer who would utilize CUDA. The policies which we have demonstrated present a policy engineer with examples as how to write logging behavior, execution limiting, and missing code completion.

Performance analysis was performed on CUDA code to justify design decisions of `eGASP`. We have found that inserting code through inline, the method that `eGASP` uses to enforce policies, is faster to execute than executing two separate CUDA functions. Also, we have found that our decision of inlining policies caused no significant performance penalties, compared to placing function calls to policy code.

We have presented a list of features that we wish to add on to `eGASP`. Through that feature list we have argued for the necessity of an open-source compiler for CUDA. As we have demonstrated throughout this thesis, it is enforcing security policies on a GPGPU was made possible through aspect-oriented programming. It is anticipated that through the results of this work, we could provide a long term goal for researching security policies and applying those security policies towards parallel and distributed computing.

REFERENCES

[1] Bader AlBassam. GH0S1R33P0R/eGASP: enforcing GPUs through Aspect Style Programming. https://github.com/GH0S1R33P0R/eGASP, 2016.

[2] Evan Andersen. How Nvidia breaks Chrome Incognito. https://charliehorse55.wordpress.com/2016/01/09/how-nvidia-breaks-chrome-incognito, 2016 (accessed February 09, 2016).

[3] Blaise Barney. Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/, 2016 (accessed May 08, 2016).

[4] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–314. ACM, 2005.

[5] Jonathan Beard. A short intro to stream processing. http://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html, 2015 (accessed February 15, 2016).

[6] Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Jingyue Wu, Xuetian Weng, Artem Belevich, and Robert Hundt. GPUCC: An Open-Source GPGPU Compiler. http://images.nvidia.com/events/sc15/pdfs/SC5105-open-source-cuda-compiler.pdf, 2015 (accessed March 25, 2016).

[7] Michael Boyer. CUDA Kernel Overhead. http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html, (accessed April 14, 2016).

[8] Robert Clucas and Stephen Levitt. CAPP: A C++ aspect-oriented based framework for parallel programming with OpenCL. In *Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*. ACM, 2015.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001.

[10] Matteo DellÁmico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana de Olivera, and Yves Roudier. HiPoLDS: A security policy language for distributed systems. In *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*, volume 7322, pages 97–112. Springer, 2012.

[11] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[12] Andrea Fornaia, Christian Napoli, Giuseppe Pappalardo, and Emiliano Tramontana. An AO system for OO–GPU programming. In *Workshop "From Object to Agents" (WOA15)*, volume 1382, pages 24–31, 2015.

[13] The Eclipse Foundation. The AspectJ project. http://www.eclipse.org/aspectj, (accessed April 14, 2016).

[14] Simon Godik, Anne Anderson, Bill Parducci, P Humenn, and S Vajjhala. Oasis eXtensible Access Control 2 Markup Language (XACML) 3. Technical report, OASIS, 2002.

[15] Intel. Intel Core i7-6700K processor (8m cache, up to 4.20 GHz) specifications. http://ark.intel.com/products/88200/Intel-Core-i7-6700T-Processor-8M-Cache-up-to-3_60-GHz, 2016 (accessed February 17, 2016).

[16] Intel. Intel Xeon Phi Product Family: Product Brief. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html, (accessed May 08, 2016).

[17] Lalana Kagal, T Finin, and A Joshi. Rei: A policy specification language. http://rei.umbc.edu, 2005 (accessed February 8, 2016).

[18] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

[19] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[20] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Symposium on Security and Privacy*, pages 175–187. IEEE, 1997.

[21] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[22] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[23] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[25] B Medeiros, R Silva, and JL Sobral. Gaspar: a compositional aspect-oriented approach for cluster applications. *Concurrency and Computation: Practice and Experience*, 2015.

[26] Nvidia. What is GPU computing? high-performance computing. http://www.nvidia.com/object/what-is-gpu-computing.html, 2007 (accessed February 15, 2016).

[27] Nvidia. programming guide::cuda toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide, 2015 (accessed February 15, 2016).

[28] Nvidia. GeForce GTX Titan X specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications, 2015 (accessed February 17, 2016).

[29] Jonathan Passerat-Palmbach, Pierre Schweitzer, Jonathan Caux, Pridi Siregar, Claude Mazel, and David R.C. Hill. Harnessing Aspect Oriented Programming on GPU: Application to Warp–Level Parallelism (WLP). *International Journal of Computer Aided Engineering and Technology*, 7(2):158–175, 2015.

[30] Daniel Reynaud. GPU Powered Malware. In *Ruxcon*, Sydney, Australia, November 2008.

[31] Dennis Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, July 1978.

[32] Phil Savage. ESEA release malware into public client, forcing users to farm Bitcoins [updated]. http://www.pcgamer.com/esea-accidentally-release-malware-into-public-client-causing-users-to-farm-bitcoins, 2013 (accessed February 18, 2016).

[33] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[34] M Sloman, N Dulay, and B Nuseibeh. SecPol: Specification and analysis of security policy for distributed systems. http://www-dse.doc.ic.ac.uk/Projects/secpol/SecPol-overview.html, 1998 (accessed February 8, 2016).

[35] Morris Sloman and Kevin Twidle. Domains: A framework for structuring management policy. In *System Objects, SIGOIS Bulletin*, pages 171–184, 1992.

[36] Richard M. Stallman. The C Preprocessor. https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html, 2001 (accessed April 12, 2016).

[37] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[38] Mahesh V. Tripunitara and Eugene H. Spafford. Security policy communication in a distributed network element. In *International Conference on Advanced Communication Technology*. ICACT, 1999.

[39] Wei-Tek Tsai, Xinxin Liu, and Yinong Chen. Distributed policy specification and enforcement in service-oriented business systems. In *IEEE International Conference on e-Business Engineering*, pages 10–17. IEEE, 2005.

[40] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder2: A policy system for autonomous pervasive environments. In *Fifth International Conference On Autonomic and Autonomous Systems*, pages 330–335. IEEE, 2009.

[41] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. GPU-assisted malware. *International Journal of Information Security*, 14(3):289–297, 2015.

[42] Mingliang Wang and Manish Parashar. Object-oriented stream programming using aspects. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2010.