THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF ELECTRICAL ENGINEERING

COMPILER-GENERATED INPUT VALIDATION VULNERABILITY ANALYSIS

MICHAEL BETTS
FALL 2015

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Electrical Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

Daniel Patrick Koller, Ph.D.
Senior Research Associate Applied Research Lab
Thesis Supervisor

Jeffrey Scott Mayer, Ph.D.
Associate Professor Electrical Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

As the usage of modern computer systems increases and more and more people delve into the art of programming, the impetus to increase the operating speed of programs has increased. One of the ways that this desire to increase program speed has affected programmers is the proliferation and increasing complexity of optimization options available in compilers. A compiler is a program that turns the human-readable pieces of code that developers write into the binary instructions that a computer runs. The existence of compiler programs allows code to run on different machines and speeds up the writing of code. When an optimizer is added to a compiler, the compiler will attempt to create machine code that the machine can run more efficiently than the code that would otherwise be generated by a literal translation of the programming code. As the sophistication of these optimizers increases, poorly written code that contains errors that would not otherwise hamper the program's output can be turned into significant security holes. These vulnerabilities have affected many pieces of code, including the Linux kernel. This project aims to begin an examination of how these errors are created, and the programming logic that can result in these security holes.

TABLE OF CONTENTS

LIST OF FIGURES iv

ACKNOWLEDGEMENTS v

Chapter 1 Introduction 1

Motivation 1
Goals and Objectives 2
Objectives 2

Chapter 2 Development of Code Snippets 3

Selection of Area of Interest 3
Undefined Behavior 4
Initial Creation of Snippets 5
Continuing Snippet Development 6

Chapter 3 Code Snippet Analysis 8

"Vanilla" Compilation Snippet Analysis 8
Creation and Reduction of Alternative Compiles 9
Classification of Alternatives 11
Benign 11
Dangerous 12
Possibly Dangerous 12
Determination of Exact Causes of Change 12

Chapter 4 Results 14

Identification of Individual Flags Causing Input Validation Vulnerabilities 14
Identification of Groups of Flags Causing Input Validation Vulnerabilities 15
Discovery of Optimizations Not Controlled by Flags 16

Chapter 5 Further Research 17

Continuing Analysis of Code Snippets 17
Security-conscious Compiler Development 18

Chapter 6 Conclusions 20

Appendix A Example code map: Addition2.cpp 21

Appendix B Example Table of Single-Flags Results: Addition2.cpp 25

Appendix C Bash Script to generate differing compilations 27

Appendix D Example Diff: Addition1.cpp vanilla and –O2 31

BIBLIOGRAPHY 36

LIST OF FIGURES

Figure 1: Exerpt from a "code mapping" 9

Figure 2: Exerpt from a "diff" file 10

ACKNOWLEDGEMENTS

I would like to thank my co-workers Kenrick Dacumos and Michael Hohnka of the Applied Research Laboratory at The Pennsylvania State University who worked with me on this project for their encouragement and support for my work. Their help in talking through some of the tough decisions on how to focus my research enabled me to proceed in my work. I would like to thank my supervisor, Dan Koller, for lending his support throughout the course of this project, and guiding me as I came up with the topic for my thesis. I would also like to thank the Applied Research Lab for providing the opportunity and the resources to complete this thesis. I would also like to acknowledge my honors advisor, Jeffrey Mayer, for his guidance throughout my time in the Penn State Schreyer Honors College. Finally, I would like to extend special thanks to my family, who have provided much support for me throughout the entire process of this thesis.

Chapter 1

Introduction

Motivation

Our modern world is relying more and more on computer systems in every part of our life. The increased reliance on, and demand for, these systems to do more and more complex tasks has led to the near necessity for engineers to be able to write computer programs to control these systems, and to allow more complex tasks to be achieved. As more people begin to have the skills to program at a basic level, the interest in and use of compiler optimizations is increasing. Compiler optimizations are important, as they take inefficient and inexpertly written code, and try to improve its operating speed when converting the written code to machine-readable code. These optimizations, while useful, can have negative consequences.

The biggest consequence is that code not written perfectly to the technical standard, that may do what its author intended if translated without any optimizations enabled, will sometimes cause large problems when compiled with an optimized compiler. These errors are non-obvious to even experienced programmers and occur in many open-source projects.

While this has been a known problem for many years, it is only recently that the increasing prevalence of these errors has caused researchers to begin studying the negative consequences of using optimized compilers. This problem was brought to our attention by two papers, *Undefined Behavior: What Happened to My Code?* and *Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior.* These papers analyzed the problems

associated with optimized compilers interacting with the C/C++ construction of undefined behavior and tried to develop tools to help coders fix their code so that these problems will not occur.

Goals and Objectives

The goal of this project is to increase the understanding of the effect of compiler optimizations on pieces of C and C++ code that has behavior undefined in the standard. This will be done by analyzing the specific options that can cause vulnerabilities to be induced by a chosen compiler, in this case the GNU Compiler Collection, GCC. Because of the time scale of the project, the analysis will be limited to a subset of the types of code that can be created.

Objectives

- Select a subset of coding operations to analyze for the effect of undefined behavior.
- Create a broad array of examples of this type of code by modifying the implementation of specific operations.
- Compile these examples with varying sets of optimizations enabled, and analyze the generated computer code to investigate the effects of specific compiler optimization options.
- Develop an understanding of what flags can cause unwanted dangerous code, and why these flags cause these problems.

Chapter 2

Development of Code Snippets

Selection of Area of Interest

This project by its very nature encompasses a broad array of topics and a vast amount of data. To test how a compiler works in its entirety, I needed to either completely test every possible piece of code with every single compilation option available, or make a complete audit of the compilers' code itself, and all the permutations of the code that can be run. Since there are several hundred different options to choose from in GCC, I needed a method to limit the scope of my part of the project into one that I could conceivably accomplish in the allotted time span.

The first way I limited the scope of the project was by dividing the sections of coding to be covered. I agreed on the various common elements of a program, such as receiving input or copying data in memory, and then chose an area with which to start. I chose to look at the obtaining and validation of input, for which I mainly looked at integer inputs. In the context of input validation, an error means that the value of the input is within selected boundaries. For example, if the program requires a positive integer less than 100, then an input validation error has occurred if the program uses a negative value, or a value greater than 100.

The second decision was one of how to limit the compilers and compiler options I was using. Because of the limited timescale I had to write my thesis, I decided to limit my coverage to the most popular open-source compiler, GCC. GCC has several hundred options in its

compiler, the number of which fluctuates by version. It also has some flags that are groupings of individual flags. These flag groups, often called, “Optimization Levels”, are suggested as standard ways to implement a specific type of optimization. Some individual flags generate warnings without effecting code, or are for languages other than C/C++. After eliminating flags that were not of interest, I generated an initial list of all the individual options, as well as the commonly used groupings of options to experiment on.

Undefined Behavior

One of the major steps to prepare for the creation of code snippets was to increase my understanding of the subject of undefined behavior. According to the C Standard, section 3.4.3, the definition of undefined behavior is, “behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.” To clarify this point, the standard notes that, “Possible undefined behavior ranges from ignoring the situation completely with unpredictable results… to terminating a translation or execution.”

In practical terms, this means that for a program to execute reliably it needs to contain no instance of undefined behavior whatsoever. This is harder than it sounds as many of the standard pieces of coding used by programmers contain undefined behavior, and yet this deficiency may in practice go unnoticed for a long time. In many cases this can lead to situations that may not result in security issues when the program is used as intended by the programmer but can create many opportunities for a hacker to exploit.

Since the standard imposes no restrictions on the processing of code that contains undefined behavior, the default treatment of the code is often to run the code in a way similar to what the one who programmed it would expect. It is only when more complicated features of optimized compilers encounter undefined behavior that the compiler can make certain error-causing decisions. A common way this occurs is that the part of the compiler that identifies code that can never be run called “dead” code. Unfortunately, this part of the compiler will remove pieces of code that can trigger only if the code is undefined. This tends to destroy naïve pieces of code specifically written to deal with edge cases, or even specific instances of undefined behavior. This and other similar parts of the compiler can nevertheless still be useful for helping to speed the execution of perfectly standard-compliant code.

Initial Creation of Snippets

In order to test how the optimizations could affect compiled code, I needed to either obtain or create some code to compile. I created several small snippets of code that I could use to test how the compiler would react to certain specific pieces of code. I used information combined from several sources to develop the code snippets. I also employed a general knowledge of the C++ programming language to write the snippets. The concepts described in the papers on undefined behavior and the examples that they provided helped to dictate the organization of the code snippets into categories based on the tests. The papers and further research also helped me to get a grasp on the appropriate complexity level of my code snippets. Another resource I used was information gained from searches online for common pieces of code used by everyday programmers.

I started the development process by deciding on a test that a programmer might do – for example, a check of whether or not two inputs were both positive followed by a check of whether or not their addition would overflow, that is, to reach a value requiring more bits than the computer could hold in the allocated memory space. I began with specific formulations present in the papers as well as common constructs taught as a part of elementary programming. I then supplemented these constructs by searching for and considering various ways to accomplish these tasks. I then checked in online forums to see that these tests were reasonable compared to the checks I found in code online.

I then slightly modified each of the elements in the list of ways to accomplish a specific check. I did this to vary the complexity of the snippets. I would add complexity by replacing static expressions in the code with variables, or by adding statements to other parts of the code to allow a possible change in how the compiler generated the code, and thus how the compiler interacted with the subject sections of code. My goal was to create snippets where undefined behavior had snuck in over multiple lines of code, or where it had occurred before a value was stored to a variable. I designed these changes to create a sliding scale of code complexity, to find the tipping point where the complexity of the code was sufficient that the compiler was not sophisticated enough to recognize that undefined behavior was being invoked, and changing the nature of the code.

Continuing Snippet Development

Upon compilation and early analysis, I realized that some of my initial code snippets were more complicated than the ones I was trying to copy from my sources, and this extra

sophistication was causing many of the snippets to fail to produce compiler-generated vulnerabilities. Only in the least complex snippets were the instances of undefined behavior exploited in optimizations that resulted in vulnerabilities. Because of this analysis, I realized that I needed to reduce the complexity of these snippets to produce the information I was looking for.

A method of reducing complexity that I tried was revising my code snippets that relied on multiple variable inputs to create versions that had one input and a relevant constant. This was in reaction to the fact that the example code snippets from the motivating papers included several examples where there was only a single variable. I created these derivative snippets only for the original snippets that had not increased in complexity but had still generated no results.

Chapter 3

Code Snippet Analysis

"Vanilla" Compilation Snippet Analysis

The first step in analyzing any snippet was to create a compile that I could consider as the baseline. This needed to be a compilation where I could correlate the original and compiled code and be reasonably sure the generated code was a faithful representation of the C++ code. I also needed to ensure that this compilation did not add any detectable vulnerabilities. I could then compare this compilation to later compilations to allow us to draw attention to parts of the code that changed between the two. I decided to refer to this baseline as a "vanilla" compile to distinguish it from the other compilations that I would later generate.

For GCC, creating a "vanilla" compile was obtained by doing the default compile, which is equivalent to using the –O0 flag, which disables all optimizations. This compile is a line-by-line translation of the C++ code, which allows specific lines of code in C++ to be mapped to blocks of assembly that perform exactly the specified operation. To begin my analysis I did precisely that: I created a two-column document with one column containing the assembly code of the "vanilla" compile, and the other containing the original C++ code of the snippet as well as some added notes (See Figure 1, below, and Appendix A). I used these notes to keep track of where in memory variables were stored, explained the carrying out of specific operations, and directed the reader to relevant parts of code that worked together.

<< endl;	movl $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, 4(%esp)
	movl %eax, (%esp)
	call _ZNSolsEPFRSoS_E

(Moves out of conditionals)	jmp .L3

else
if(a+b<a) (edx = a)
(eax = b)
(edx = edx + eax (a+b))
(eax = a)
(Compare eax to edx (a to a+b))
(if edx was >= eax (a+b>=a), leave conditional)
	.L4:
	movl 24(%esp), %edx
	movl 28(%esp), %eax
	addl %eax, %edx
	movl 24(%esp), %eax
	cmpl %eax, %edx
	jge .L3

cout << "The sum has overflowed."	movl $.LC2, 4(%esp)
	movl $_ZSt4cout, (%esp)
	call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc

Figure 1: Exerpt from a "code mapping"

Creation and Reduction of Alternative Compiles

To prepare for the creation of alternative compiles, I obtained the list of all GCC flags from the GCC documentation for the version of GCC being used in the project. I first removed from the list any compiler flags that were for a language other than C/C++. I then also removed the flags that generated warnings and those that did not affect the generated code itself. I also removed the options that allowed the user to change a compiler value to an arbitrary value, as testing every possible value for such options would take too much time to be useful given the limited timescale of this project. I left in the optimization level flags, which begin with an O, as these may not be individual flags, but they contain groupings of flags, and these are much more commonly used than any of the individual flags.

I then modified a Bash script originally generated by Kenrick Dacumos, a co-worker at ARL working on another part of the project, which would compile a code snippet repeatedly with the list of compiler flags I had generated (See Appendix C). In its original form it would then record compiles that generated both compile time errors and runtime errors, as well as generate a list of any compiled assembly files that differed from the original. I needed to modify this script as the original design did not accept input. I created an input file and redirected the input of the running C program to this file. I also modified this script to aid in various analyses by generating a file using the "diff" utility for the compilations that differed (See Figure 2, below, and Appendix D), as well as modifying the parameters it used to generate a compile during the later analyses that were more complex.

```
.L3:                                                    .L3:
        movl    24(%esp), %edx                  |               movl    $.LC2, 4(%esp)
        movl    28(%esp), %eax                  <
        leal    (%edx,%eax), %ebx                               leal    (%edx,%eax), %ebx
        movl    $.LC3, 4(%esp)                  <
        movl    $_ZSt4cout, (%esp)                              movl    $_ZSt4cout, (%esp)
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI           call
_ZStlsISt11char_traitsIcEERSt13basic_ostreamI
        movl    %ebx, 4(%esp)                                   movl    %ebx, 4(%esp)
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
        call    _ZNSolsEi                                       call    _ZNSolsEi
        movl    $.LC4, 4(%esp)                  |               movl    $.LC3, 4(%esp)
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
```

Figure 2: Exerpt from a "diff" file

I then took the generated code snippets and removed the snippets that showed no differences from the "vanilla" compile and prepared the remaining snippets for analysis.

Classification of Alternatives

Analysis of the generated alternatives began by using the files generated by the diff utility. These files contained the vanilla compile in one column and the generated compile in the other, with markers to denote lines that differed. Using the diff files, I was able to analyze the exact lines that changed and see what exactly changed about them. Using the analysis of the vanilla compile, I was also able to see exactly what C++ lines had had their implementation changed. On preliminary investigation, I classified the changes into the three categories shown below.

In addition to these classifications, I created short descriptions of the specific changes that the various compilation flags had created. When I understood the logic of the compiler, I explained why these changes appeared (See Appendix B for example).

Benign

The first category was “Benign,” defined as no security implications being present in the changes that the compiler flag had generated. An example of this is the “-ffreestanding” flag which, for a certain snippet, switched the storage location of two variables A and B in the stack. This did not affect the amount of memory used or the security of the program in any way, and was only a cosmetic change to the program. If no unexplained or harmful change remained, I would then categorize this flag as “Benign” for the code snippet.

Dangerous

The second category was “Dangerous,” which was used when a flag caused a change that could be positively identified to affect the security or operation of the program. This contained several flags that when researched were accomplishing their intended effect, but this effect could cripple code that was not designed with this in mind.

Possibly Dangerous

The final category was “Possibly Dangerous,” which contained any code snippet that was not overtly “Dangerous” but could not be positively identified as completely “Benign”. I eventually moved the majority of these snippets to one of the other two categories upon further examination.

Determination of Exact Causes of Change

Some of the flags used in our tests were actually groupings of various other flags into what GCC refers to as “optimization levels.” These flags were often useful in identifying when errors caused by a group of flags had a possibility of occurring. The methods used were good at identifying the effects of individual flags, but sometimes these combinations of flags would show vulnerabilities that did not appear in any of the individual flag compilations.

Whenever this occurred, it was critical to determine which of the flags in the combinations were actually causing the vulnerabilities to occur. There was a more complicated analysis used to identify what interactions were causing problems to appear. The first step was to

apply the group of flags, and remove each flag in the group one at a time to find all necessary flags. Then all the flags needed for the vulnerability to occur that I discovered were added to the "Vanilla" compile. I then analyzed this new compilation for the vulnerability. If these flags alone did not induce the vulnerability, or removing subsets of these flags from larger groups of flags did not prevent the vulnerability, I needed to do more analysis.

Techniques used to identify the effect of harder-to-identify flags involved finding a subgroup of the group that originally caused the error, adding in the known necessary flags, and compiling. This allowed me to identify if large groups of flags were responsible for the behavior. Another technique involved removing all unknown flags, and adding them in one at a time until the vulnerability occurred, and adding the flag that caused the change to the list of known problematic flags. This allowed the uncovering of some complicated combinations of flags that led to specific vulnerabilities.

Chapter 4

Results

Identification of Individual Flags Causing Input Validation Vulnerabilities

Many of the individual flags I found to be “Dangerous” were simply doing their jobs, as these flags were created to be used in special circumstances. These flags are not included in any of the optimization level flags that group many flags together and are not normally used. Examples of this category of flags are the flags –ftrapv, and –fprofile-generate.

-ftrapv serves a purpose that causes direct problems in the proper operation of code not designed for it. It detects when an operation containing an integer will overflow, and throws an exception. This changes the behavior of the C/C++ code, and can break code, but should be used only if the program was designed for this option.

-fprofile-generate is an example of the other kind of flag that was “Dangerous”, but not especially interesting. It creates files while it is being run that detail the inner workings of the code. This flag is useful when doing in-house testing or preparing to use the data generated to create a better-optimized compile using the –fprofile-use flag.

There was only one flag, of the ones that were “Dangerous”, that exhibited the behavior for which I was looking. This flag was the flag –fstrict-overflow. In every case where I found a problem with the integer input operations I was attempting, this flag would repeatedly be the one that would cause problems. This flag, as the name suggests, implemented the section of the C/C++ standard that states that an integer operation that overflows is undefined, by removing any code that could only execute if integer overflow occurred. Whenever there was a single variable operating with a constant, this flag, on its own, could remove checks that prevented the

program from preforming unwanted behavior. The checks it was able to corrupt included the checks shown below. This is because it was able to deduce that the checks would always be false, even though there are cases where if these were actually evaluated, the OS would evaluate them as true.

- If(A+100<A), where the compiler knew A was positive
- If((A*100)/100!=A)
- If(abs(a)<0)

Identification of Groups of Flags Causing Input Validation Vulnerabilities

Many of the code snippets showed interesting changes when compiled with the optimization-level flags, especially the –O2 flag. For some of these, the changes that occurred were not seen when any of the individual flags were applied. This was important as it showed there was some important byplay occurring in the combination of various flags. The following checks were effected by what turned out to be the same grouping of flags.

- If(A+B<0) where the compiler knew A and B were positive
- If(A/B<0) where the compiler knew A and B were negative

This grouping of flags contained flags within the –O2 and –O1 level. The specific set of flags needed was –fstrict-overflow and –ftree-vrp and either –ftree-fre or –ftree-dominator-opts, as well as something else that was not a flag contained in the –O1 optimization flag grouping.

Discovery of Optimizations Not Controlled by Flags

One of the most interesting elements of GCC that I uncovered in my research was that flags did not necessarily control all of the optimizations in GCC. As I have already stated the necessary path to remove several of the specific checks was a set of specific flags, but in finding this set of flags I also found that something else was needed that was not controlled by its own flag, but could only be turned on by an optimization level group of other flags.

I discovered this hidden optimization when I noticed that there was a difference in the compilation whenever I was trying to determine the flags that caused the check condition if(A+B<0) to be removed when the compiler could deduce that both of the variables A and B were positive. As stated above, I had discovered the fact that the necessary optimizations were some subset of the flags denoted as –O1, and the two flags –fstrict-overflow and –ftree-vrp, which were contained in –O2. One of the steps I took to try to determine which of the –O1 flags were needed was to replace –O1 with what the documentation listed as its constituent parts, 38 specific flags. The problem was that the 38 flags turned on individually caused the compiler to generate different code than the code that was generated by turning them all on with the –O1 flag.

To test what was happening I compared compilations where I turned nothing on, with compilations where I turned on –O1 then manually removed everything listed as a part of that flag. These two compilations had differences, and this result caused me to be able to more easily find certain results, and have led to some of my plans for further research.

Chapter 5

Further Research

Continuing Analysis of Code Snippets

As the original subject of this research is very broad indeed, there are many ways this project can and will be continued. My work on input validation was partly an experiment and test of some of the concepts and methods that I can use to increase the efficiency of and value gained from creating and analyzing different types of code. By avoiding the taking of paths that I found were dead ends, and by using some of the understanding of various compiler portions I gained, I can increase the speed of both creation and analysis of code snippets. I am also planning to take advantage of some of the things I found inherent in the compiler, such as optimizations without specific flags, to suggest new combinations of flags including the various optimization level flags, to see the effect of these uncontrollable optimizations.

In addition to increasing the breadth of the study through different types of code, another important effort I plan to continue my research into is examining other popular compilers to see how readily the results I gain can translate to other compilers. This might allow us to see whether there are specific compilers that are better at avoiding the creation of undefined behavior-related bugs.

A third method of continuing the analysis of code snippets is to increase the depth of my search. This research treated the compiler as a “black box” to try to gain some intuition into the inner workings of the compiler. By also investigating the available source code of the compiler

and the files generated during the intermediate stages of the compilation, I can perhaps learn in a more comprehensive manner the compiler's process of making undesired decisions.

Security-conscious Compiler Development

In addition to the continuing the research on compilers, another extension to this project will come in the development of tools to mitigate the effects of harmful compilations on security. There are three methods that could be used to achieve this goal.

The first method would involve simply creating a listing of the "Dangerous" compiler optimization flags that I have found for each compiler. I would then find sets of flags that negate the dangerous side effects of optimizations without crippling the optimization process overmuch. This has the advantage of being very simple and easily applied, but as the optimizations present in every compiler change and expand with newer versions it would quickly become obsolete.

A second, more involved method would be to create our own compile flag or flags that implement the same functionality as above but with a finer-toothed comb in each compiler. These flags would manipulate the compiler code and processes to avoid optimizing certain code constructs. This would be both much more complicated and much more time consuming, as this would require a deep understanding of both the code that is being affected and the optimization processes of each compiler. It would also limit our results to only open-source compilers like GCC, as proprietary commercial compilers such as Microsoft's Visual C++ compiler prevent us from changing them.

A third option to apply our findings is to create our own compiler that would act similarly to the above flag optimizers. Using what I have found about optimizations I could create a compiler that strayed away from dangerous optimizations, or only applied these optimizations in safe areas. The major downside is that this compiler would have to be created from scratch, and as such would not have many of the features of other compilers that have large teams working on them. On the other hand, I would have complete control of this compiler, and thus it could not update and destroy our work by changing the way the compiler works.

As these methods all have upsides and downsides, the final solution may be some combination of these options, and there is a significant research left before I can attempt a comprehensive audit of a compiler.

Chapter 6

Conclusions

The work done on this subject is only beginning to scratch the surface on how compiler optimizations can affect the security of code. A large portion of this project was simply determining how to approach this subject in a useful manner, as the amount of data that I generated was enormous. Since this is a relatively new subject, I had to create the tools I used and a large portion of the analysis was done by hand. This allowed us to gain insight into the operation of the compiler, and to begin to get an understanding of the compiler decisions that were being made that caused unwanted behavior.

The work done in this thesis, while useful, is focused on a very small fraction of the area that could and should be covered and is nowhere near a complete picture of a compiler. Much more research needs to be done before comprehensive conclusions can be drawn.

Appendix A

Example code map: Addition2.cpp

```
int main()                                  main:
{                                           .LFB971:
                                                    .cfi_startproc
                                                    pushl   %ebp
                                                    .cfi_def_cfa_offset 8
                                                    .cfi_offset 5, -8
                                                    movl    %esp, %ebp
                                                    .cfi_def_cfa_register 5
                                                    pushl   %ebx
                                                    andl    $-16, %esp
                                                    subl    $32, %esp
                                                    .cfi_offset 3, -12

cout << "Please enter 2 integers."                  movl    $.LC0, 4(%esp)
                                                    movl    $_ZSt4cout, (%esp)
                                                    call
                                                    _ZStlsISt11char_traitsIcEERSt13basic_os
                                            treamIcT_ES5_PKc

<< endl;                                            movl
                                                    $_ZSt4endlIcSt11char_traitsIcEERSt13b
                                            asic_ostreamIT_T0_ES6_, 4(%esp)
                                                    movl    %eax, (%esp)
                                                    call    _ZNSolsEPFRSoS_E

int a,b; cin >> a (stores a in 24(%esp))            leal    24(%esp), %eax
                                                    movl    %eax, 4(%esp)
                                                    movl    $_ZSt3cin, (%esp)
                                                    call    _ZNSirsERi

>> b; (stores b in 28(%esp))                        leal    28(%esp), %edx
                                                    movl    %edx, 4(%esp)
                                                    movl    %eax, (%esp)
                                                    call    _ZNSirsERi
```

```
if(a<0) (eax=a)                                  movl    24(%esp), %eax
(bitwise and eax and eax)                        testl   %eax, %eax
(jumps to else if the above not negative)        jns     .L2

cout << a                                        movl    24(%esp), %eax
                                                 movl    %eax, 4(%esp)
                                                 movl    $_ZSt4cout, (%esp)
                                                 call    _ZNSolsEi

<< " is less than 0."                            movl    $.LC1, 4(%esp)
                                                 movl    %eax, (%esp)
                                                 call
                                                 _ZStlsISt11char_traitsIcEERSt13basic_os
                                                 treamIcT_ES5_PKc

<< endl;                                         movl
                                                 $_ZSt4endlIcSt11char_traitsIcEERSt13b
                                                 asic_ostreamIT_T0_ES6_, 4(%esp)
                                                 movl    %eax, (%esp)
(Moves out of conditionals)                      call    _ZNSolsEPFRSoS_E
                                                 jmp     .L3
else                                             .L2:
if(b<0) (eax=b)                                  movl    28(%esp), %eax
(bitwise and eax and eax)                        testl   %eax, %eax
(jumps to else if the above not negative)        jns     .L4

cout << a                                        movl    28(%esp), %eax
                                                 movl    %eax, 4(%esp)
                                                 movl    $_ZSt4cout, (%esp)
                                                 call    _ZNSolsEi

<< " is less than 0."                            movl    $.LC1, 4(%esp)
                                                 movl    %eax, (%esp)
                                                 call
                                                 _ZStlsISt11char_traitsIcEERSt13basic_os
                                                 treamIcT_ES5_PKc

<< endl;                                         movl
                                                 $_ZSt4endlIcSt11char_traitsIcEERSt13b
                                                 asic_ostreamIT_T0_ES6_, 4(%esp)
                                                 movl    %eax, (%esp)
                                                 call    _ZNSolsEPFRSoS_E
                                                 jmp     .L3
```

else	.L4:
if(a+b<a) (edx = a)	movl 24(%esp), %edx
(eax = b)	movl 28(%esp), %eax
(edx = edx + eax (a+b))	addl %eax, %edx
(eax = a)	movl 24(%esp), %eax
(Compare eax to edx (a to a+b))	cmpl %eax, %edx
(if edx was >= eax (a+b>=a), leave conditional)	jge .L3

cout << "The sum has overflowed."	movl $.LC2, 4(%esp)
	movl $_ZSt4cout, (%esp)
	call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc

<< endl;	movl $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, 4(%esp)
	movl %eax, (%esp)
	call _ZNSolsEPFRSoS_E

(end of conditionals)	.L3:

(Calculate A+B)(edx = a)
(eax = b)
(ebx = edx + eax)
	movl 24(%esp), %edx
	movl 28(%esp), %eax
	leal (%edx,%eax), %ebx

cout << "The sum is "	movl $.LC3, 4(%esp)
	movl $_ZSt4cout, (%esp)
	call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc

<< a+b (grab a+b from above)
(use return from above to continue writing)
(output an integer)
	movl %ebx, 4(%esp)
	movl %eax, (%esp)
	call _ZNSolsEi

<< "."	movl $.LC4, 4(%esp)
	movl %eax, (%esp)
	call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc

```
<< endl;                                    movl
                                            $_ZSt4endlIcSt11char_traitsIcEERSt13b
                                        asic_ostreamIT_T0_ES6_, 4(%esp)
                                            movl    %eax, (%esp)
                                            call    _ZNSolsEPFRSoS_E

return 0;                                   movl    $0, %eax
                                            movl    -4(%ebp), %ebx
                                            leave
```

Appendix B

Example Table of Single-Flags Results: Addition2.cpp

Flag	*Category*	*Description*
-fcrossjumping	Benign	added jump to remove duplicating code above and below .L2
-fdata_sections	Benign	added section headers
-ffreestanding	Benign	Switched storage location of a and b on the stack
-ffunction-sections	Benign	added function headers
-finhibit-size-directives	Benign	removed assembly level size directives
-fmerge-all-constants	Benign	changed section headers
-fmerge-constants	Benign	changed section headers
-fno-ident	Benign	removed assembly level ident directive
-freorder-blocks-and-partitions	Benign	generates ".section .text.unlikely" areas with no code or data throughout program
-fsection-anchors	Benign	moves ".local _ZStL8__ioinit" and ".comm _ZStL8__ioinit,1,1"
-fshrink-wrap	Benign	renamed label .L6 to .L7
-fsplit-stack	Benign	breaks the stack up into multiple small sections
-fstack-check	Benign	adds code to check if esp has been modified
-fverbose-asm	Benign	adds assembly level comments
-fvisibility-ms-compat	Benign	adds hidden field to main header
-fstack-protector-all	Benign	Adds an area to the stack using a value offset by 20 from the gs register gs:20, and stores it in the stack before the program is run. Afterwards it compares the two and makes sure they are still equal. If gs:20 is not equal to what is in the location it was stored in, It causes a function to signal that the stack check failed.
-fkeep-inline functions	Benign	adds all function calls code to the program.
-O (-O1)	Benign	Avoids reloading A and B and calls cout with less instructions
-O2	Benign	-O1 plus reordering of code sections to reduce code space and # of conditional jumps on various branches.
-O3	Benign	identical to -O2 for this code
-Ofast	Benign	identical to -O2 for this code
-finstrument-functions	Possibly Dangerous	added code to top and bottom of functions
-flto	Possibly Dangerous	Added special strings for link time optimization
-fomit-frame-pointer	Possibly Dangerous	avoids push/pop of ebp where it is unessesary
-Og	Possibly Dangerous	-O with changed label names and added code outside of main
-Os	Possibly Dangerous	Uses stack wherever possible and uses push and pop over moves. (increased stack usage could ease the ability of interfering with programs by gaining stack access, also could increase ease of stack overflow.)

Flag	*Category*	*Description*
-fpic (-fPIC -fpie -fPIE)	Possibly Dangerous	made all function calls/jumps relative with thunk table instead of absolute addressing (should only be used for libraray functions)
-mrtd	Dangerous	assumes called functions pop stack causing stack to grow with each function call if not properly handled. (nonstandard flag used for such a purpose)
-ftrapv	Dangerous	Library functions are used for addition that cause code to break on overflow. (Similarly used when this behavior is desired)
-frecord-gcc-switches	Dangerous	adds details of compiler flags used as well as other options (records information on compile that could be used to attack programs)
-fprofile-generate	Dangerous	creates code to generate profile to improve later compiling(records information when program is run about program paths most used, this leaves information for an attacker)

Appendix C

Bash Script to generate differing compilations

Automate.sh

```
#!/bin/bash

#Author:      Kenrick M Dacumos
#Date:        05/26/2015
#Description: This program will automate the manual_test_1 test in
order to allow us to
#             recompile the code snippets (.cpp) with each of the
interesting gcc
#             gcc compiler options.
#Inputs:      None
#Outputs:     .s file of each recompiled code snippet
#             .o file of each recompiled code snippet
#             .ii file of each recompiled code snippet
#             a.out file of each recompiled code snippet
#             Text file containing any errors produced from running
the a.out executables
#             Text file containing a list of compiler options that
produced a compiler
#               error.  Note:  We are ignoring any compiler warnings.
#             Text file contiaining a list of compiler options that
produce a variation
#               in the original and the new .s files.
#Instructions:  To use this program you must do the following:
#                   1. Run and compile your original code snippets and
save all output
#                      files produced from the compiler (-save-temps)
#                   2. Obtain .txt file containing the list of gcc
compiler options to
#                      test.  Each compiler option in the .txt file
should be sepearted
#                      by a newline character.
#                          3. Change the "Initialze variables" to
point towards all of the
#                      approprate directories
#                   4. Run this program.  Conduct each different run
of this program in
#                      a seperate folder in order to seperate
different tests from each
```

```
#                       other.  (Example Folder names: test_2, test_3,
etc.)
#                       Note:  For organization purposes, please keep
all files you need
#                              for any one test in its respective test
folder.  For
#                              example, keep a copy of this program in
each of your test
#                              folders (test_1, test_2, etc.)
##########################################################################
###################

#Initialize variables
#**************************************************************************
******************
dir_home='/home/michael/Desktop/Testing_add1/GCC'
cd $dir_home  #Change working directory to dir_home

dir_code_snip='original_code_snippets/cpp_snippets'  #The location of
the original .cpp files
dir_vnla_compile_files='original_code_snippets/vcompiled_files'  #The
location of the vanilla compiled files(.s, .o, .ii, a.out)
dir_recompile_files='test_files/recompiled_files'  #The location of
the recompiled files (.s, .o, .ii, a.out)
compiler_options_list='test_files/options_list_newline.txt' #The path
to the list of gcc compiler options to test.  Should be a '\n'
seperated file.
options_that_vary_assem='test_files/options_that_vary_assem.txt'  #The
path to the list of gcc compiler options that vary the assembly files
from the original
#**************************************************************************
******************




#Functions
#**************************************************************************
******************
function compileRun {
      #Compile
      file_ext='.cpp'
      g++ -save-temps $option $dir_code_snip'/'$codesnippet  #Compile
with option. Save all output files
      if [ $? -eq 0 ]  #Check for any compiler errors
      then
            echo "NO compile errors"
```

```
            filename=${codesnippet%$file_ext}  #Remove .cpp file
extension from the codesnippet variable

      dir_compiler_output_files=$dir_recompile_files'/'$filename'/'$opt
ion
            echo $dir_compiler_output_files
            mkdir $dir_compiler_output_files  #Create a directory with
the name of the option to hold the compiler output files
            mv $filename'.s' $filename'.o' $filename'.ii' a.out
$dir_compiler_output_files  #move compiler output files to its
appropriate directory
      else
            echo $option >> test_files/compile_error_options.txt
#append option to text file
      fi
      #Run
      $dir_compiler_output_files/a.out < "input.txt"
      if [ $? -eq 0 ]  #Check for any run errors
      then
            echo "NO run errors"
      else
            echo $option >> test_files/run_error_options.txt  #append
option to text file
      fi
}

function diff_fcn {
      file1=$dir_vnla_compile_files'/'$filename'/'$filename'.s'
      file2=$dir_compiler_output_files'/'$filename'.s'
      diff $file1 $file2
      if [ $? -ne 0 ]
      then
            echo "Option: "$option  "Code Snippet: "$filename'.cpp' >>
test_files/options_that_vary_assem.txt
            diff $file1 $file2 -y >
$dir_compiler_output_files'/diff.txt'
      fi
}

#*****************************************************************
******************




#Main Program
#*****************************************************************
******************
```

```
FILES=$dir_code_snip'/*'
#$FILES must be a variable reference to a filepath.
#If it is the actual filepath itself it will not iterate over each
file as it is supposed
#to.

for codesnippet in $FILES #work on all files in the directory
specified by FILES
do
      codesnippet=${codesnippet#$dir_code_snip'/'}
      echo ********************$codesnippet**************************
      echo $codesnippet
      while read option
      do
            #echo $option $codesnippet
            compileRun
            diff_fcn
      done <$compiler_options_list
done

#*********************************************************************
*******************
```

Appendix D

Example Diff: Addition1.cpp vanilla and –O2

```
        .file   "Addition1.cpp"                                 .file   "Addition1.cpp"
        .local  _ZStL8__ioinit                  |               .section .rodata.str1.1,"aMS",@progbits,1
        .comm   _ZStL8__ioinit,1,1              <
        .section .rodata                        <
.LC0:                                                   .LC0:
        .string "Please enter 2 integers."                      .string "Please enter 2 integers."
.LC1:                                                   .LC1:
        .string " is less than 0."                              .string " is less than 0."
.LC2:                                                   .LC2:
        .string "The sum has overflowed."       <
.LC3:                                           <
        .string "The sum is "                                   .string "The sum is "
.LC4:                                           |       .LC3:
        .string "."                                             .string "."
        .text                                   |               .section .text.startup,"ax",@progbits
                                                >               .p2align 4,,15
        .globl  main                                            .globl  main
        .type   main, @function                                 .type   main, @function
main:                                                   main:
.LFB971:                                        |       .LFB998:
        .cfi_startproc                                          .cfi_startproc
        pushl   %ebp                                            pushl   %ebp
        .cfi_def_cfa_offset 8                                   .cfi_def_cfa_offset 8
        .cfi_offset 5, -8                                       .cfi_offset 5, -8
        movl    %esp, %ebp                                      movl    %esp, %ebp
        .cfi_def_cfa_register 5                                 .cfi_def_cfa_register 5
        pushl   %ebx                                            pushl   %ebx
        andl    $-16, %esp                                      andl    $-16, %esp
        subl    $32, %esp                                       subl    $32, %esp
        .cfi_offset 3, -12                                      .cfi_offset 3, -12
        movl    $.LC0, 4(%esp)                                  movl    $.LC0, 4(%esp)
        movl    $_ZSt4cout, (%esp)                              movl    $_ZSt4cout, (%esp)
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI           call
    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI
        movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ost <
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
        call    _ZNSolsEPFRSoS_E                |               call
    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostr
        leal    24(%esp), %eax                                  leal    24(%esp), %eax
        movl    %eax, 4(%esp)                                   movl    %eax, 4(%esp)
        movl    $_ZSt3cin, (%esp)                               movl    $_ZSt3cin, (%esp)
        call    _ZNSirsERi                                      call    _ZNSirsERi
        leal    28(%esp), %edx                                  leal    28(%esp), %edx
```

```
        movl    %edx, 4(%esp)                                   movl    %edx, 4(%esp)
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
        call    _ZNSirsERi                                      call    _ZNSirsERi
        movl    24(%esp), %eax                  <
        testl   %eax, %eax                      <
        jns     .L2                             <
        movl    24(%esp), %eax                  <
        movl    %eax, 4(%esp)                   <
        movl    $_ZSt4cout, (%esp)              <
        call    _ZNSolsEi                       <
        movl    $.LC1, 4(%esp)                  <
        movl    %eax, (%esp)                    <
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI <
        movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ost <
        movl    %eax, (%esp)                    <
        call    _ZNSolsEPFRSoS_E                <
        jmp     .L3                             <
.L2:                                            <
        movl    28(%esp), %eax                  <
        testl   %eax, %eax                      <
        jns     .L4                             <
        movl    28(%esp), %eax                  <
        movl    %eax, 4(%esp)                   <
        movl    $_ZSt4cout, (%esp)              <
        call    _ZNSolsEi                       <
        movl    $.LC1, 4(%esp)                  <
        movl    %eax, (%esp)                    <
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI <
        movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ost <
        movl    %eax, (%esp)                    <
        call    _ZNSolsEPFRSoS_E                <
        jmp     .L3                             <
.L4:                                            <
        movl    24(%esp), %edx                                  movl    24(%esp), %edx
                                                >               testl   %edx, %edx
                                                >               js      .L8
        movl    28(%esp), %eax                                  movl    28(%esp), %eax
        addl    %edx, %eax                      <
        testl   %eax, %eax                                      testl   %eax, %eax
        jns     .L3                             |               js      .L9
        movl    $.LC2, 4(%esp)                  <
        movl    $_ZSt4cout, (%esp)              <
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI <
        movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ost <
        movl    %eax, (%esp)                    <
        call    _ZNSolsEPFRSoS_E                <
.L3:                                                    .L3:
        movl    24(%esp), %edx                  |               movl    $.LC2, 4(%esp)
        movl    28(%esp), %eax                  <
        leal    (%edx,%eax), %ebx                               leal    (%edx,%eax), %ebx
```

```
        movl    $.LC3, 4(%esp)                  <
        movl    $_ZSt4cout, (%esp)                              movl    $_ZSt4cout, (%esp)
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI           call
    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI
        movl    %ebx, 4(%esp)                                   movl    %ebx, 4(%esp)
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
        call    _ZNSolsEi                                       call    _ZNSolsEi
        movl    $.LC4, 4(%esp)          |                       movl    $.LC3, 4(%esp)
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI           call
    _ZStlsISt11char_traitsIcEERSt13basic_ostreamI
        movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ost <
        movl    %eax, (%esp)                                    movl    %eax, (%esp)
        call    _ZNSolsEPFRSoS_E        |                       call
    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostr
        movl    $0, %eax                |                       xorl    %eax, %eax
        movl    -4(%ebp), %ebx                                  movl    -4(%ebp), %ebx
        leave                                                   leave
                                        >                       .cfi_remember_state
        .cfi_restore 5                                          .cfi_restore 5
        .cfi_restore 3                                          .cfi_restore 3
        .cfi_def_cfa 4, 4                                       .cfi_def_cfa 4, 4
        ret                                                     ret
                                        >       .L8:
                                        >               .cfi_restore_state
                                        >               movl    %edx, 4(%esp)
                                        >       .L6:
                                        >               movl    $_ZSt4cout, (%esp)
                                        >               call    _ZNSolsEi
                                        >               movl    $.LC1, 4(%esp)
                                        >               movl    %eax, (%esp)
                                        >               call
        _ZStlsISt11char_traitsIcEERSt13basic_ostreamI
                                        >               movl    %eax, (%esp)
                                        >               call
        _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostr
                                        >               movl    24(%esp), %edx
                                        >               movl    28(%esp), %eax
                                        >               jmp     .L3
                                        >       .L9:
                                        >               movl    %eax, 4(%esp)
                                        >               jmp     .L6
        .cfi_endproc                                            .cfi_endproc
.LFE971:                                |       .LFE998:
        .size   main, .-main                                    .size   main, .-main
        .type   _Z41__static_initialization_and_destruction_0 | .p2align 4,,15
_Z41__static_initialization_and_destruction_0ii:|               .type   _GLOBAL__sub_I_main,
@function
.LFB980:                                |       _GLOBAL__sub_I_main:
                                        >       .LFB1008:
```

```
        .cfi_startproc                                          .cfi_startproc
        pushl   %ebp                            |               subl    $28, %esp
        .cfi_def_cfa_offset 8                   |               .cfi_def_cfa_offset 32
        .cfi_offset 5, -8                       <
        movl    %esp, %ebp                      <
        .cfi_def_cfa_register 5                 <
        subl    $24, %esp                       <
        cmpl    $1, 8(%ebp)                     <
        jne     .L6                             <
        cmpl    $65535, 12(%ebp)                <
        jne     .L6                             <
        movl    $_ZStL8__ioinit, (%esp)                 movl    $_ZStL8__ioinit, (%esp)
        call    _ZNSt8ios_base4InitC1Ev                 call    _ZNSt8ios_base4InitC1Ev
        movl    $__dso_handle, 8(%esp)                  movl    $__dso_handle, 8(%esp)
        movl    $_ZStL8__ioinit, 4(%esp)                movl    $_ZStL8__ioinit, 4(%esp)
        movl    $_ZNSt8ios_base4InitD1Ev, (%esp)        movl    $_ZNSt8ios_base4InitD1Ev,
(%esp)
        call    __cxa_atexit                            call    __cxa_atexit
.L6:                                    |               addl    $28, %esp
        leave                           |               .cfi_def_cfa_offset 4
        .cfi_restore 5                  <
        .cfi_def_cfa 4, 4               <
        ret                                             ret
        .cfi_endproc                                    .cfi_endproc
.LFE980:                                |       .LFE1008:
        .size   _Z41__static_initialization_and_destruction_0 <
        .type   _GLOBAL__sub_I_main, @function          <
_GLOBAL__sub_I_main:                    <
.LFB981:                                <
        .cfi_startproc                  <
        pushl   %ebp                    <
        .cfi_def_cfa_offset 8           <
        .cfi_offset 5, -8               <
        movl    %esp, %ebp              <
        .cfi_def_cfa_register 5         <
        subl    $24, %esp               <
        movl    $65535, 4(%esp)         <
        movl    $1, (%esp)              <
        call    _Z41__static_initialization_and_destruction_0 <
        leave                           <
        .cfi_restore 5                  <
        .cfi_def_cfa 4, 4               <
        ret                             <
        .cfi_endproc                    <
.LFE981:                                <
        .size   _GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main      .size
        _GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main
        .section .init_array,"aw"               .section .init_array,"aw"
        .align 4                                        .align 4
        .long   _GLOBAL__sub_I_main                     .long   _GLOBAL__sub_I_main
```

```
                                                >               .local  _ZStL8__ioinit
                                                >               .comm   _ZStL8__ioinit,1,1
        .hidden __dso_handle                                    .hidden __dso_handle
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"      .ident  "GCC: (Ubuntu 4.8.4-
2ubuntu1~14.04) 4.8.4"
        .section .note.GNU-stack,"",@progbits                   .section .note.GNU-stack,"",@progbits
```

BIBLIOGRAPHY

Xi Wang , Haogang Chen , Alvin Cheung , Zhihao Jia , Nickolai Zeldovich , M. Frans Kaashoek, Undefined Behavior: What Happened to My Code?, Proceedings of the Asia-Pacific Workshop on Systems, July 23-24, 2012, Seoul, Republic of Korea [doi>10.1145/2349896.2349905]

Xi Wang , Nickolai Zeldovich , M. Frans Kaashoek , Armando Solar-Lezama, Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior, Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, November 03-06, 2013, Farminton, Pennsylvania [doi>10.1145/2517349.2522728]

Lattner, Chris. "LLVM Project Blog." What Every C Programmer Should Know About Undefined Behavior #1/3. N.p., 13 May 2011. Web. 14 Nov. 2015. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

J. Regehr. A Guide to Undefined Behavior in C and C++, July 2010. http://blog.regehr.org/archives/213

International Orginazation for Standardization (ISO). (2011) ISO 9899:2011 Information technology -- Programming languages -- C . Geneva, Switzerland.

ACADEMIC VITA

Academic Vita of Michael Betts
mbetts360@gmail.com

Education
Major: Electrical Engineering
Honors: Electrical Engineering

Thesis Title: Compiler-Generated Input Validation Vulnerability Analysis
Thesis Supervisor:

Work Experience
Date: 5/21/14 - present
Title: Distinguished Undergraduate Researcher
Description:
Institution/Company: Applied Research Lab (ARL) at Penn State
Supervisor's Name: Michael Hohnka

Skills and Certifications
Active Government Secret Clearance
Certified LabVIEW Associate Developer
Cisco Certified Entry Network Technician (CCENT)
CompTIA A+ Computer Technician certification
Understanding of programming in C++, Java, and MIPS

Grants Received:
H. Thomas and Dorothy Willits Hallowell Scholars Endowment through Through College of Engineering
H. Smyser Bair Memorial Scholarship through Schreyer Honors College
Academic Excellence Scholarship through Schreyer Honors College
Professional Engineers in Private Practice Scholarship
Alumni Memorial Scholarship

Awards: None

Professional Memberships: None

Publications: None

Presentations: None

Community Service Involvement: Oakwood Presbyterian Church

International Education (including service-learning abroad): None

Language Proficiency: English