## Abstract

This project is to analysis the mechanism of GS stack protection in Microsoft's compiler framework. To investigate the usage of GS stack protection and the performance of the software products generated, we believe that a more clear and specific relation can be found between the two, which may shed light on improving the efficiency and security of the compiler.

In this project, several fields, including compiling, buffer overflows and stack protection, will be studied, in order to get a deep understanding of the performance issue of stack protection mechanisms. It is normal to think that the gain of security is offset by the loss in efficiency. However, according to Tim Burrell from Microsoft Security Engineering Center (MSEC) and my supervisor Daniel Page, there is some sign showing that the performance of GS protected program may be improved somehow. The OpenSSL is used to confirm the existence of that phenomenon.

Then, to figure out why the GS stack protection mechanism gives rise to improvement in performance or not, we have investigated how the GS protection mechanism of the Visual Studio compiler works and come up with five hypothesises to explain the phenomenon theoretically. Then, experiments and analysis are conducted to verify those hypothesise. In the progress, we can develop a more clear view of the performance issue in stack protection mechanisms and contribute to the stack protection technology.

During the whole project, the main achievements are:

- We have surveyed the techniques in buffer overflow attacks and their countermeasures in Chapter 2.
- We have looked into Microsoft's stack protection technique GS and conducted experiments to evaluate the performance of the OpenSSL with GS protection in Chapter 3.
- We have listed five hypothesises to explain how techniques like GS really affect the performance of the outcome programs. One hypothesis, memory paging, has been emphasized and validated through experiments and analysis in the part 2 of Chapter 4.

# Table of Contents

# Chapter 1. Introduction

## 1.1 Introduction

The security of software involves a billion-dollar industry, which means that the attackers can cause huge loss to victims by attacking holes in software programs, and users are willing to pay large sum of money to protect against attacks.

One of the most common vulnerabilities is the buffer overflow. In applications and operating systems, instructions are kept in the memory, where attackers can manage to modify them or crash the programs. The IT industry has been suffering from attacks on the buffer for more than 20 years. In 1988, the Morris Worm downed up to 10% of machines on the Internet and cost millions of dollars, by taking advantage of a buffer overflow in the finger service [1]. Since then, the buffer overflow attacks have been developed with an increasing notoriety. In 2003, the Blaster Worm plagued computers throughout the Internet. This worm exploited a buffer overflow on the Microsoft operating systems [2]. By checking the Microsoft Security Bulletin [3], you can find historic and current vulnerabilities in Microsoft's products. Within the long list, quite a number are related to buffer overflows. Actually a large percentage of attacks are based on the buffer overflows, as the following table shows.

Table 1 Advisories according to [4]

| Year | Number of Advisories | Number of Adv. Relating to buffer overflows |
|---|---|---|
| 2004 | 9 | 7 |
| 2003 | 28 | 18 |
| 2002 | 37 | 21 |
| 2001 | 37 | 19 |
| Total | 111 | 65 |

The stack buffer overflow, as its name indicates, is part of this general approach. Attacks exploiting stack buffer overflows are popular in attackers' communities. Since the computer architectures which most programs are compiled and run on have the same stacks, the stack frame is often targeted.

Since a large amount of attacks are on the stack of programs, the stack protection becomes a critical issue of software security. There are various stack shielding technologies, one example of which is to add extra instructions during the compilation phase of a program. StackGuard, StackShield and Stack-Smashing Protector (SSP) for GCC, /GS for Microsoft, are of that type. These techniques all basically use prologues to add cookies or canaries into stack when a function is called, and let epilogues to check those codes when the function returns. They differ from each other in details, because they are working with different compilers.

It is normal to think that any gain in security is offset by a loss in efficiency. By investigating the use of GS stack protection, we believe that a clear and specific relation can be found between the security and performance of the generated software. This may be helpful to improving the future generations of the stack protection technology.

In this project, the above fields, including compiling, buffer overflows and stack protection, will be covered, in order to check whether it is true or not that the gain of security is considered to be offset by the loss in efficiency, because extra data is added to the original programs. To figure out whether the GS stack protection mechanism gives rise to improvement in performance or not, we have to investigate how the compiler framework works. So, some concrete cases must be designed in order to test the above assumption. By comparing the cases with and without the GS option, we can develop a more clear view and contribute to the stack protection technology.

## 1.2 Aims and objectives

Generally speaking, this project is to analysis the mechanism of GS stack protection used by Microsoft's compiler framework. It is normal to think that the gain of security is offset by the loss in efficiency. However, according to Tim Burrell from Microsoft Security Engineering Center (MSEC) and my supervisor Daniel Page, there is some case showing that the performance of GS protected program may be improved somehow. To investigate the usage of GS stack protection and the performance of the software products generated, we believe that a more clear and specific relation can be found between the two, which may shed light on improving the efficiency and security of the compiler.

In pursuit of this general goal, the following objectives have been achieved:

- Survey the techniques in buffer overflow attacks and their countermeasures.
- Investigate Microsoft's stack protection technique GS.
- Conduct experiments to evaluate the usage of stack protection techniques and the performance of the software programs generated.
- Analyse results derived from experiments and figure out how the techniques like GS really affect the performance of the outcome programs.

## 1.3 Organization

This dissertation is divided into five chapters.

At first, this chapter 1 is an introduction about the motivation and objectives of our research.

Then, chapter 2 talks about the background knowledge in fields of programming functions, the stack, buffer overflows and protection techniques.

Chapter 3 looks into the GS stack protection and its performance issue. In this part, the OpenSSL is used as a case-study, showing that the GS protection may improve the performance in certain situations.

Chapter 4 develops hypothesises from five aspects trying to explain the counter-intuitive phenomenon described by Chapter 3. Research and experiments are conducted to verify them, especially the hypothesis about memory paging.

Finally, in chapter 5, a conclusion is made and the future work is also discussed.

# Chapter 2. Background

## 2.1 The stack and functions

In order to describe what stack-based buffer overflow attack is, we start by showing the architecture of the computer. There are various types of architectures with different features, but most of them are consist of processors and memory devices. The processor operates the data on the memory. The software programs are just a set of instructions telling the processor how to deal with data. Programming upon this kind of architecture needs the knowledge of program and subroutines.

### 2.1.1 Subroutines

A series of operations that the processor does with the data is named a program. When one task must be executed in different parts of a program, subroutines are needed to avoid redundancy. A subroutine is a piece of program which can be stored in one special place and be called to undertake a special task by the main program. So, the subroutine part need to be coded only once and called by other programs multiple times. [5]

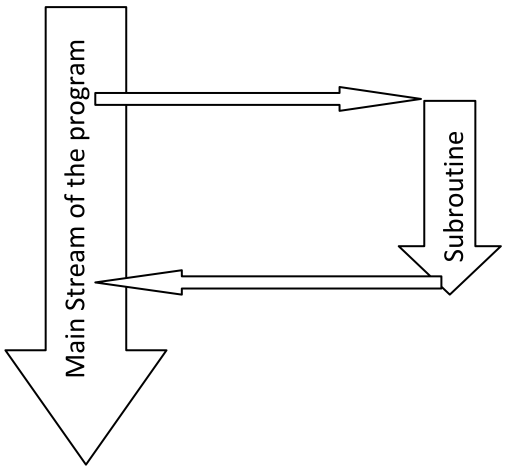


Figure 1 Subroutine

The basic subroutine needs an entrance and an exit [6], so it can be called and return afterward. To use subroutine is actually about the methodology of software programming. We need to reuse many code segments and sometimes, for example, want a function to call itself recursively. But on the low level of the most basic operations, the processor cannot recur itself. Considering what we might use to transform recursion to non-recursion code in C, the concept of stack shall be introduced.

### 2.1.2 Memory layout and stack

When a program is executed, it has to store data in memory. In the modern computer architectures, the memory layout of programs, such as C and C++ programs, is normally of the following segments: static data segment, heap and stack. The static data segment is to hold globe and static variables, initialized and uninitialized. Then, the heap is used for dynamic memory allocation, while the stack region is used in the same manner of the data structure stack, last-in-first-out.

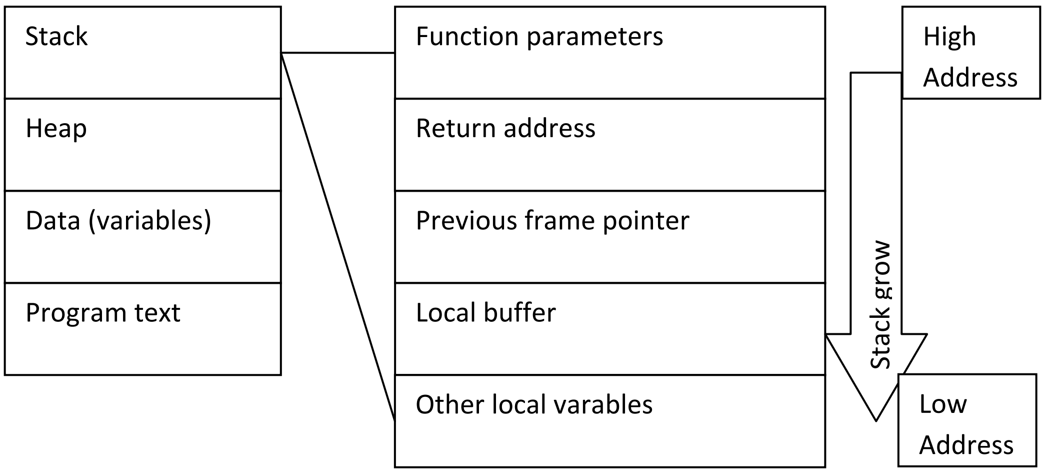


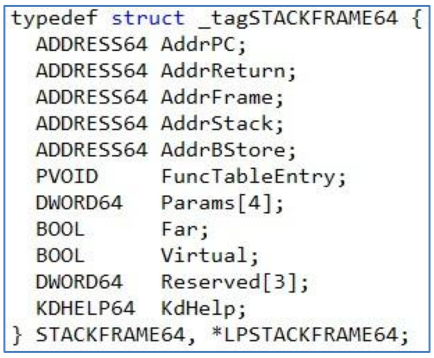
Figure 2 Stack in memory according to [7]

To store the return address of one subroutine, we need one register. When multiple subroutines are calling each other, we need more than one register to retain the return addresses. In this situation, a stack is required.

The stack is a kind of data structure which handles data in the last in first out way. In memory, a stack is an area which handles data in that manner. To deal with the multiple subroutines situation, the call stack, or namely run-time stack, can pop the latest return address when the most recently called function finishes. Then, the processor lets the Stack Pointer (SP) point to the next register in the stack.

As in functions, there are things other than return addresses, such as local variables, return states, pointers, and structures. All of those should be in the same context of the return addresses, so the stacks also have to store the data. As in the IA32 architecture, the stack grows from the high address to the low address, opposite to the direction of heap. In Unix-like systems, the stack has the responsibility of handling function calls. A stack frame is allocated for each call, with which things such as local variables, return values, and return addresses, are stored.

Although the contents are much the same in general, what precisely the stack frame contains is determined by the compiler. The following figure from MSDN shows the stack frame under Microsoft’s compiler framework [8].



```
typedef struct _tagSTACKFRAME64 {
  ADDRESS64 AddrPC;
  ADDRESS64 AddrReturn;
  ADDRESS64 AddrFrame;
  ADDRESS64 AddrStack;
  ADDRESS64 AddrBStore;
  PVOID     FuncTableEntry;
  DWORD64   Params[4];
  BOOL      Far;
  BOOL      Virtual;
  DWORD64   Reserved[3];
  KDHELP64  KdHelp;
} STACKFRAME64, *LPSTACKFRAME64;
```

Figure 3 Structure of stack frame. Taken from [8]

### 2.1.3 Instructions for manipulating the stack

Different processor families provide different instruction set. The x86 instruction set architectures, which are based on Intel 8086 processors, have their own set of instruction codes to handle operations on the stack. Operating systems on such kinds of platforms provide application binary interface (ABI) between applications and the operating systems. Other processor families for embedded devices like ARM and MIPS support embedded application binary interface (EABI) on operating systems.

To manipulate the stacks, there are two basic operation instructions: pop and push. Processor registers of CPU, which store references of data for CPU accessing fast, are encoded in the instruction sets. So, the operation instructions can operate on those registers with ease. The following codes might be useful:

EAX: This is the accumulator register.

EBP: This is the base pointer.

ESP: This is the stack pointer.

EIP: This is the next instruction address.

RET: This is return code, marking the end of subroutine and back to the previous program.

NOP: This means no operation.

JMP: This can be used to jump to an address.

MOV: This means “move”. It is used to copy data from one address to another.

It is fairly easy to use IDEs like visual studio to generate disassemble version from high level language source code. So is it to generate machine language code from disassembly code. In machine language, the instructions are a series of numbers. An operation code (opcode) is part of those [9]. On one hand, with the help of those instructions, programmers can manipulate low level devices including registers, heaps and stacks. On the other hand, without sufficient protections, programs can be hijacked by attackers.

## 2.2 Vulnerabilities

### 2.2.1 Buffer Overflows

Software is surrounded by various attacks ever since it has been written. Buffer overflows are certainly one of the most common security holes for those attacks to take advantage of. [10] Buffer overflow vulnerabilities exist in almost everywhere, including operating systems, PC applications, and network programs, etc. By checking the Microsoft Security Bulletin [3], you can find historic and current vulnerabilities in Microsoft’s products. Within the long list, quite a number are related to buffer overflows. Through successfully attacking on buffer overflows, attackers may gain the priority of other user and even take control of the systems. It seems to be more serious when the Internet is taken into account, as any internet user may try to get control of certain remote host which has potential buffer overflow vulnerabilities [11].

In general, the buffer overflow, as its name implies, is a situation when the data being written into a buffer exceeds the size of the buffer. Without any bounds checking or protection, this causes the nearby area of buffer being overwritten and corrupted.

So the buffer overflow is normally categorized into two kinds: stack-based and heap-based buffer overflows. The heap-based buffer overflow of cause occurs in the heap area, whilst the stack-based buffer overflow happens on the stack [12].

Here in my research, I mainly focus on the stack-based buffer overflow. Overwriting the return addresses on the stack, the stack-based buffer overflow can cause malicious codes being executed [13], which leads to serious consequences.

Similar to other buffer overflows, stack-based buffer overflows write larger amount of data to a fixed size buffer, overwrite data on adjacent area, and get the stack corrupted, which cause the program to work in unexpected ways. It is considered one of the traditional security holes to be used in attackers' communities [14].

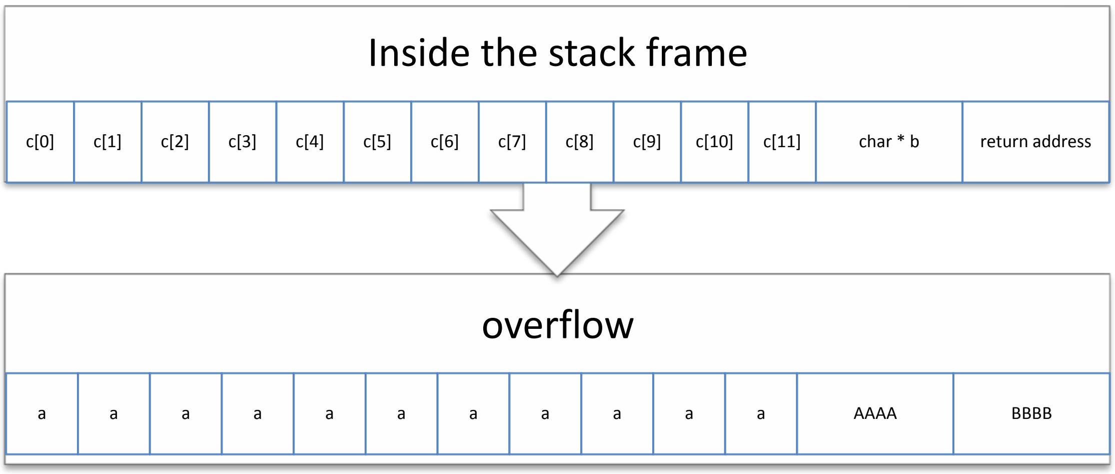


Figure 4 Stack buffer overflow

As the above graph shows, when a string "aaaaaaaaaaaaaaaaAAAABBBB" is copied into the array c, nearby area in the stack will be overwritten.

To say this vulnerability is common, it is because that the operation of inputting data to the buffer is common and two mainstream programming languages, C and C++, are lack of built-in bounds checking or protection against overflows. Take the character array for example: any write operation with unexpected large data can pass the bound of the array without explicit warning. Some other languages like Java have their own bound-checking methods, but may also face the buffer overflow problem. Since Java has to run on Java Virtual Machine (JVM) which is possibly written in C and/or C++, buffer overflows can still exist in Java.

### 2.2.2 Exploits

In a stack frame, when the function is newly called, the return address is pushed on the top of the stack. Containing the address of where to go after the function ends, the return address becomes a target of attacks. Thousands of attackers are taking advantages of the buffer overflows to attack software programs and systems. As shown in the previous part, the principle of stack-based buffer overflow attacks is of no complexity. The overflow of an array in the stack may overwrite nearby data, such as the return address. By changing the return address, an attacker can redirect the flow of the program to wherever he likes, including malicious code in the memory space. This traditional approach is called stack smashing.

This graph shows the steps and supporting techniques which an attacker might use to produce an attack.

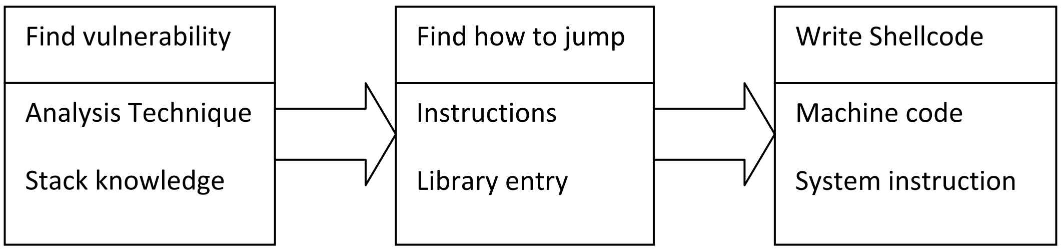


Figure 7 Workflow of attacks

Under the above idea, after having targeted a stack buffer overflow, attackers have to put suitable code somewhere in the program, and use overflows to modify any return address so the program would jump to that code. However, in order to carry out the above attacks, three conditions should be at least partially satisfied. According to [15], these conditions are:

1) Instructions of system calls should be known,
2) Useful library entry points should be known, and
3) The stack placement should be known.

Although those instructions and libraries vary from one system version to another, many of them are open for public development use.

The attacker can only start to design his attack after knowing necessary instructions, libraries, and stack layout. Then, he could know what operation code should be used to jump, how to find the entry of the program, what tricks can be used to find the return address, and which system calls should be revoked, etc. The answers to those questions are not always the same, since those instructions and libraries are sometimes changed.

In real life, writing exploits also requires knowledge of disassemble. There are some widely-used disassemblers, like OllyDbg, IDA and WinDbg. Among them, OllyDbg and WinDbg are free.

#### 2.2.2.1 Shellcode

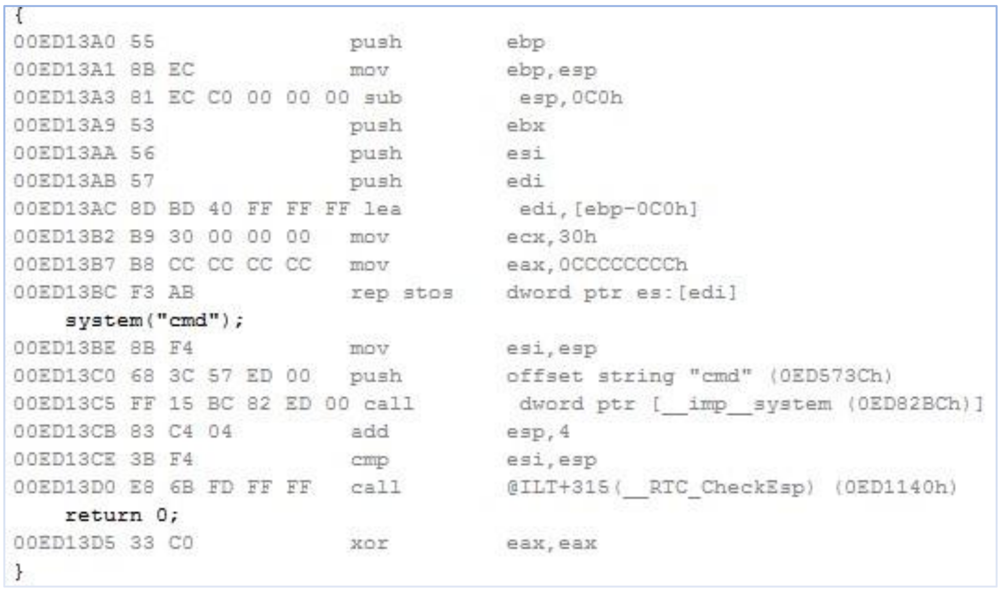
It is easy to arrange suitable code in the buffer, either to inject it or to use existing code there. To inject suitable code, one can place the code on the stack, on the heap, or in the static data segment. As in the most common way, shellcode [16] [17], attackers inject codes in the form of machine code as payload in the space. If there exist some code the attacker can use, he could just jump to that sentence in a straightforward way.

The code that actually performs attacks on certain program is the shellcode. [13, 18-19] It is normally written in machine code. Because attackers typically use those codes to revoke the command shell for testing, it gets that name.

The code to open a command line window in Windows is like this:

```
int main(int argc, char * argv[]){
        system("cmd");
        return 0;}
```

Then, with the help of Visual Studio, we can see the disassembly and code bytes version:



```
{
00ED13A0 55                push        ebp
00ED13A1 8B EC             mov         ebp,esp
00ED13A3 81 EC C0 00 00 00 sub          esp,0C0h
00ED13A9 53                push        ebx
00ED13AA 56                push        esi
00ED13AB 57                push        edi
00ED13AC 8D BD 40 FF FF FF lea          edi,[ebp-0C0h]
00ED13B2 B9 30 00 00 00    mov         ecx,30h
00ED13B7 B8 CC CC CC CC    mov         eax,0CCCCCCCCh
00ED13BC F3 AB             rep stos    dword ptr es:[edi]
    system("cmd");
00ED13BE 8B F4             mov         esi,esp
00ED13C0 68 3C 57 ED 00    push        offset string "cmd" (0ED573Ch)
00ED13C5 FF 15 BC 82 ED 00 call         dword ptr [__imp__system (0ED82BCh)]
00ED13CB 83 C4 04          add         esp,4
00ED13CE 3B F4             cmp         esi,esp
00ED13D0 E8 6B FD FF FF    call        @ILT+315(__RTC_CheckEsp) (0ED1140h)
    return 0;
00ED13D5 33 C0             xor         eax,eax
}
```

Figure 6 Disassembly with source code and code bytes

Then, theoretically the bytes code is copied down as shellcode string. But in real life it should be optimized to fit into certain exploits. The string is like:

char shellcode[]={0x00,0xED,0x13,0xA0,0x55,…,0x6B,0xFD,0xFF,0xFF};

#### 2.2.2.2 Jump to shellcode

Then, the harder thing is to cause the program to jump to that code. It is exactly in this part that we need to overflow a buffer. The return address attack is the most common one. When larger length of data is copied into the array, the data will overflow to where it is anticipated, normally the return point. If the buffer is corrupted, the attacker can take the control of the flow of the program. Then, the attacker can use various instructions to let the execution flow go into his malicious code. The following figure illustrates this situation.

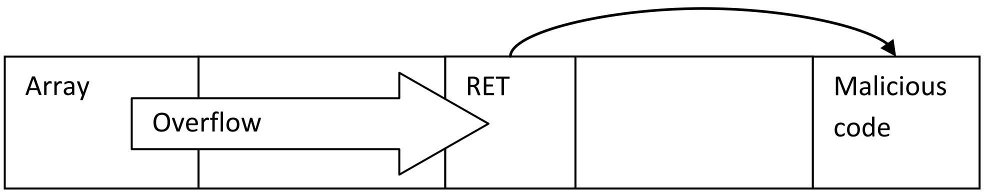


Figure 7 Jump to malicious code

It is a key point on writing an exploit to lead the program to the shellcode. Although differences between environments can be serious barriers to exploits, various techniques are applied to overcome that. The NOP sled is a widely-used one. As introduced previously, the NOP is the no-operation instruction. Then, the NOP sled technique is to place a number of NOP instructions before the shellcode, resulting in the attacked program executing through those NOPs and reaching the shellcode. Since long sequences of NOPs are too obvious, normal trivial codes (like xor eax,eax) are used [20].

However, the shellcode can sometimes be hard to reach by just sliding down. In this case, we can try to reach some points where we can jump to the shellcode. The “jump to register” technique is widely used to let the attacked program jump to a certain register, such as EIP. Since there is little restriction on “JMP” instruction, it can be used to jump to various places for execution of shellcode.

Besides the basic stuff talked above, attackers have been developing new approaches all the time. In languages like Java and C#, there is some mechanism checking the bounds of arrays, while in some external libraries of C and C++, they have bounds checking functions. However, attackers can still get away with their malicious code by changing the exception mechanism. This is called the exception-handler hijacking. [21] By using this method, attackers can bypass the Microsoft Windows Structured Exception Handling (SEH) mechanism. In Windows, the information of exception handling forms a linked list on the stack. These SEH records on the stack can be overwritten and it might be a serious exploit.

## 2.3 Protections

### 2.3.3 Detect security holes

In order to protect the program against attacks, what comes first is to find out where the vulnerabilities are. It is somehow more fatal and difficult than working out how to deal with them, because sometimes the source code is too complex for us to determine whether it is safe or not.

There are many approaches to detect security holes. To find buffer overflows, we can use ways including language-based methods [22], dynamic detectors [23-25] [7, 26] and static analysis [27-29].

#### 2.3.3.1 Static Analysis

Source code is what every program comes from. To detect security holes through analysing source code is called static analysis. It performs checking and analyzing prior to the run time of a program. Static analysis is commonly used when the source code is available. Even if the source code is not available, analyzers can still perform static analysis by using techniques like disassembling and reverse engineering.

Basically, manual auditing is one kind of static analysis, but it is inefficient. Compared to manual audits, analysis tools are always welcome. With the help of static analysis tools, programmers can check their code much faster and more frequently, without being knowledgeable on vulnerabilities. “Aim for good, not perfect” [27], a program can never be secure enough, and the result of static analysis tools should be always evaluated by human beings. We will rely a lot on tools, but still have to bear in mind that the static analysis is only part of our software development life cycle.

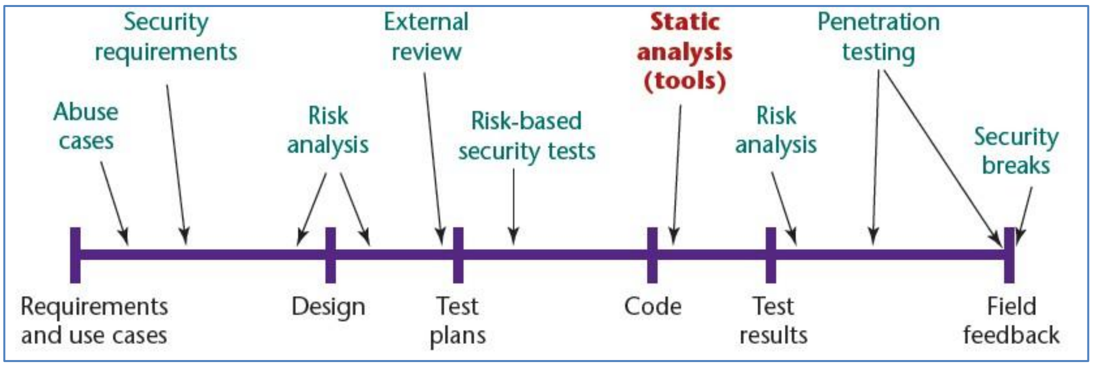


Figure 8 Static analysis in software development lifecycle. Taken from [27].

Three types of static analysis tools are normally used: annotation driven tools, symbolic analysis tools, and model-based tools.[30] Annotation driven tools, like CSSV [31] and Splint [32], use annotations in an inter-procedural phrase. They are inefficient to large programs. Symbolic analysis tools, like ARCHER [33], use information databases to trace variables and the status of programs. Model-based tools exact a model from the source code and analyze it. CodeSurfer [30] is of the third kind.

The architecture of those static analysis tools is normally consist of the constraint-generator and the error-detector [30]. The constraint-generator will generate rules from the source code, and then the error-detector will detect code lines that break the rules. For example, suppose that a C program contains the following instructions:

*char buffer[10];*

*buffer[10] = ‘a’;*

An analysis tool will produce the diagnostic message:

*Suspicious line: out-of-bounds*

*buffer[10] = ‘a’;*

*Constraint: index in array buffer should be less than 10*

*Error: buffer[10]: 10>=10*

which indicates there is a potential buffer overflow in this line. At first, the constraint-generator goes through the source code and lists constraints like this:

Table 2 Constraints

| Line Number | Constraint | Constraint Number |
|---|---|---|
| 4 | Index in array buffer <10,>=0 | #1 |
| 8 | Index in array buffer2 <20,>=0 | #2 |
| … | … | … |
| 50 | Index in array bufferN < 50, >=0 | #n |

When the error-detector is looking for potential overflows, it will mark the places where risky variables are involved. If possible, another table could be generated to show potential errors:

Table 3 Errors

| Line Number | Variable | Constraint Number |
|---|---|---|
| 25 | buffer2 | #2 |
| 45 | buffer1 | #1 |
| … | … | … |
| 70 | bufferN | #n |

Having looked at various tools, a general scheme of static analysis can be divided to three steps:

A. Scan the whole program.

B. Store the information of the program, including the structure, variables, etc.

C. Allocate suspicious vulnerabilities.

**First step**: This is not limited to source code scanning. The Unix program “grep” may be able to do scanning, but grep is far from suitable. In order to get necessary information from the program, we need syntax analysis as well as lexical analysis. This part makes it more like a compiler. Actually, in this project, the compiler is one of the important things that we cannot miss.

**Second step**: During this step, the information derived from first step can be stored in various forms, like abstract syntax tree (AST) [27]. Using AST is a compiler-based way. Flowcharts are also used for special algorithms to process [28]. To record those call stacks, the calling context tree (CCT) which retains a method in each node and takes the entry point as the root, is also used [34]. This CCT data structure can be helpful in analysing the stacks of programs.

**Third step**: In order to allocate buffer overflows, some analysis algorithms must be employed. The CodeSurfer [30] uses precise pointer analysis algorithms, which let some pointers point to the buffer and observe their values. In short, the information from previous step is processed.

Moreover, the static analysis methods do not guarantee the correctness, since they do not trace how the program is running. But they still provide ways to show the possible situation of buffer overflows.

#### 2.3.3.2 Dynamic Analysis

Dynamic analysis is performed in the run time of programs. In order to get useful results, programs shall be tested with sufficient data in dynamic analysis. This technique is normal in debugging programs.

Dynamic analysis on stack buffer overflows mainly focus on the bound checking and out-of-bound pointer issues. To monitor the behaviours of programs in dynamic analysis, the normal way is to add various tags in the source code before running the programs. Inserting instrumentation at compile time is one method. The other typical one is to wrap the binary executable directly. No matter which it is, this may lead to heavy overhead. [26]

When the buffer overflow occurs, the states of the program may be corrupted and later error report may result in incorrectness. In case of this happens during the dynamic analysis, the program state should be protected. One method is to stop the overwrite operation and report errors before corruption happens.

Take WinSafe [7] for example. The WinSafe is using binary rewriting method. It will take Windows portable executables as input, find functions in the disassembly of the programs, and modify them using binary instrumentation. The main idea of WinSafe’s binary instrumentation is to set up cannaries when functions start, and check them when functions are returned. After finding corruptions of return address and previous frame pointer, if canary values are not intact the WinSafe notifies existence of buffer overflow vulnerability and writes details of buffer overflow to the log file.

This method of analysis is used widely. Different names of canaries are given in different tools. For example, this extra data is called “cookie” in Microsoft’s GS, which will be talked later.

The following figure shows the mechanism of canary setting and canary checking in the context of one function. Before the function is actually executed, we name this phrase the function prologue. After the function body is executed, it is named the function epilogue. So, in those two moments, the dynamic analysis tools may add canaries and check them. It is no doubt that there is performance penalty in that. We may refer to this in the performance issue part.

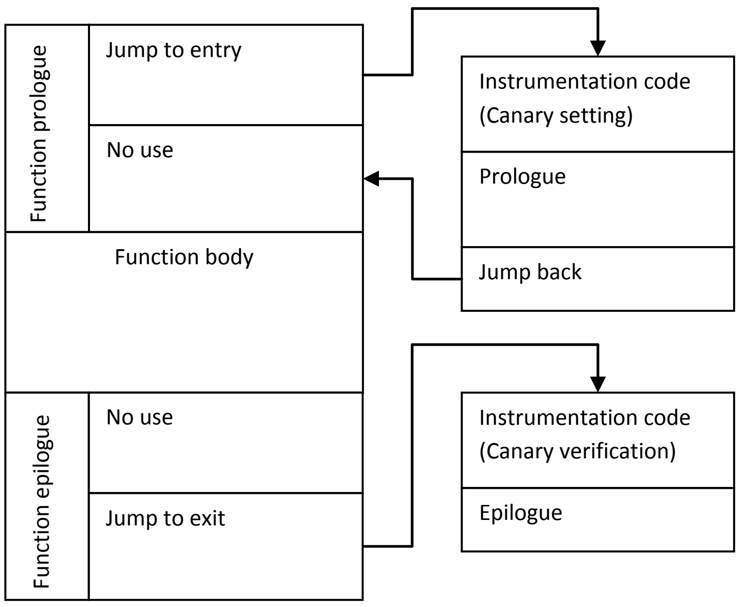


Figure 9 Dynamic analysis of a function According to [7]

Dynamic analysis methods during compiling time have to face two problems: library functions and memory blocks. In the first problem, many library functions in C were written years ago and they are lack of protection mechanism in themselves. The normal way to solve this problem is to build new library functions which can replace the old ones. In the second problem, arrays and structs are considered as single blocks in memory, so buffer overflows within those areas are hard for compile-based methods to detect and protect. This problem is related to the C standard. As those problems exist, it is difficult to generate suitable test suites to cover all the situations as well as to derive distinct output [35].

### 2.3.4 Countermeasures

After the detection of holes, we have to fix them. Vulnerabilities are keeping being patched, while the techniques of exploiting are becoming more and more sophisticated. Although it is easy to fix holes individually and never too late to mend them, we still need general and systematic ways to protect programs against buffer overflows.

According to the two sub-goals of attacking buffer overflows, defence methods should generally focus on two aspects: forbidding buffer overflows, and preventing the execution of unexpected code. As to the first case, efforts are made to change the coding style and remove those potentially risky pieces. OpenBSD used this method by manually modifying the source code to ensure bounds and pointers checking [36]. As with the second aspect, most of the protection techniques are trying to stop the execution of unauthorized operations on sensitive data. The famous StackGuard uses this strategy by adding canaries [37]. Both two aspects should be taken into consideration.

#### 2.3.4.1 Hardware-based

For the sake of copy rights, there is large interest in digital rights management (DRM) techniques. In this situation, hardware protection against security exploits becomes popular. Then, people are designing hardware protections. Some are widely used nowadays, like the non-executable data protection (NX bit) and data execution prevention (DEP).

Since the programs are executed on hardware in the lower level, techniques on hardware can decide whether a piece of code or operation can be executed or not. Generally speaking, hardware protection can protect data from being unauthorized accessed and disable certain risky operations under certain situations. However the hardware is secure, it is of no use unless being consistence with existing software standards. So the DEP is actually a set of technologies, including hardware and software [38].

Hardware-enforced DEP can mark most of the memory locations as non-executable. Attempt to run code on those memory areas will be stopped and end up with an exception. So, when this protection is applied, attackers' effort to insert executable code in the memory locations will be in vain.

The hardware-enforce DEP needs the support of CPUs. By only marking some memory address as non-executable, this technique would not work. It is the processor that decides whether certain piece of code from memory can be executed or not. So, the hardware-enforced DEP relies on CPUs, and marks memory with tags that the CPUs can recognize. Typically, the mark is made on the page table entry (PTE) of the memory page. Moreover, suitable attribute tags are marked on memory pages according to the processor architectures.

Advanced Micro Devices (AMD) and Intel have agreed to use Windows-compatible architectures. So, both processor families are compatible with DEP. The no-execute page-protection (NX) processor feature is defined by AMD, while the Execute Disable Bit (XD) feature is defined by Intel. These features are turned on by default as long as the CPUs are in Physical Address Extension (PAE) mode. Since this PAE mode is set as default, customers do not need to check it. [38]

As to prevent stack-based buffer overflow attacks, hardware-based techniques mainly focus on how to build a secure stack to store the return address. The SmashGuard in [39] saves the return address in a hardware stack. Each function call will push the return address and the stack frame pointer onto the hardware stack. This hardware stack will verify the return address every time the function is returned. If mismatch occurs, a hardware exception will be thrown. The operating systems will not ignore any hardware exception.

#### 2.3.4.2 Software-based

Many approaches are making changes to the source code in different ways. Some operate through source code transformation [40]. Some others provide solutions on operating system level. There are also ways using binary modification for buffer overflow protection[4]. We separate the various methods of BOF protection into two group according to this paper [41] : kernel-enforced and compiler-enforced protection.

The kernel is underneath the visual code and controls the environment. By modification of virtual memory space and conducting access control methods, the kernel-enforced protection is able to prevent the execution of malicious code. The memory management unit access control lists (MMU ACLs) is one of such kind methods. It only allows operations that are on the access control list access memory space, and forbids unauthorized access to the memory data. Another method is address space layout randomization (ASLR) [42] . Since some attacks are based on static values on the memory, ASLR can protect data against exploits by randomize the mapping of memory addresses. Attackers have to use brute-force ways to find binary images, library locations, and stack return address. This increases the risk of exploit failures.

As discussed in the detection part, with the help of compiler we can derive sufficient knowledge of the structure, control flow, data types of the program. So, the compiler-enforced protection is our main point. The basic idea of compiler-enforced protection is to add extra data into the code which will indicate unexpected modifications of the stack.

In compiler-based protections, StackGuard is the famous one. It works with GCC and patches canaries to the code. It provides random canaries and validates them afterwards. [37]  But this mechanism is not perfect and can be defeated. Overwriting local variables is possible to bypass StackGuard [41].

ProPolice Stack-Smashing Protection (SSP) seems to be a better protection than other compiler-based mechanisms. Besides placing canaries at the beginning and/or the end of return address, SSP tries to keep an eye on the stack.

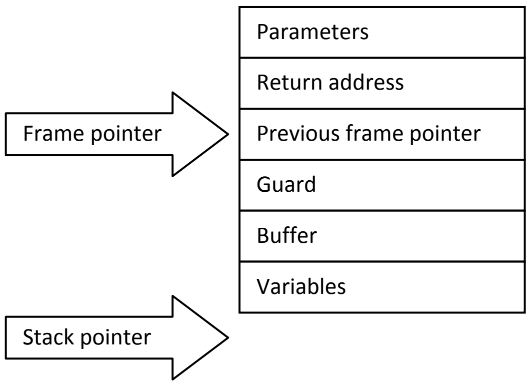


Figure 10 Placing guard According to [41]

SSP is not undefeated. Small buffer existing in the program may lead to exploits that redirect the return address to shellcode.

The techniques talked above are typically used together. The hardware and software usually are combined in real life, like the Data Execution Prevention (DEP) which is actually a set of hardware and software technologies [38].

### 2.3.5 Performance issue

The performance issue is an essential part of the mitigation technologies. In this context, the performance is the performance of the execution of protected programs. A protection with heavy overhead is impractical and useless. So we always face tradeoffs between security and performance.

According to above materials, the degradation of performance normally comes from the following factors: extra data, and extra operations. Extra data like canaries and cookies, can take up the memory space to some extent. Extra operations like checking canaries/cookies and inserting data, can slow down the execution of programs.

The general idea of optimization on this issue is to reduce unnecessary operations and data.

**One optimization is based on selective protection.** Since buffer overflows exist with buffers, Canary protection will not apply to stack frames which contain no buffer. Furthermore, if a buffer cannot be overflowed, no canary will be added, either. A more aggressive idea is to only protect functions that contain string buffers. This is because of an assumption that most buffer overflow vulnerabilities target string buffers, like char arrays. [43]

**There are also ways trying to reduce the overhead on checking the cookie or canary.** The element cache to record the frequently used referents is useful. By doing experiments with the cache size, it shows that a two-element cache provided a good balance between these tradeoffs. Compile-time optimization can provide improvement in performance, too, like loop-invariant code motion (LICM) of the referent lookup. [44]

**Splitting the stack may result in a significant solution, too.** Yves et al. claimed to present an efficient method without performance loss [45]. In their technique, the standard stack is divided to multiple stacks.

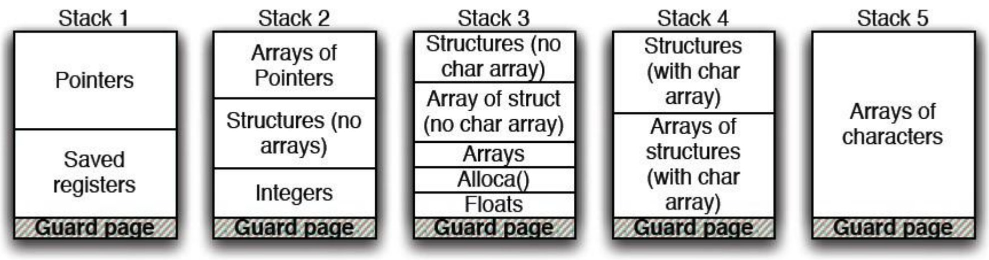


Figure 13 Five stacks. Taken from [45]

So in this case, the memory overhead is large. Compilers in this case have to allocate five stacks and calculate the gaps between those stacks. There is no extra performance overhead, because extra operations and data are made in compile time. [45]

**When it comes to GS**, efforts are made to improve its performance in every version. The main methods to decrease overhead are: choosing GS levels, and reducing unnecessary protections.

Since GS has got several versions with different overhead and security level, GS switch is applied to meet various requirements. This means that users can choose different level of GS protections according to the demands [46]. With this function, GS with wider protection scope will not be applied to places where normal GS is sufficient.

The GS do not protect every function. After classifying and identifying the functions where no GS cookie is needed or not, the performance is gain with reducing the unneeded guarding. [46]

We are looking into more details of GS stack protection in next chapter.

# Chapter 3. GS and its performance

## 3.1 GS stack protection

### 3.1.1 History & Present

Being the largest software company in the world, Microsoft's products are widely used and widely targeted by attackers. In order to fight against various attacks, Microsoft has been developing security mechanisms along with its software development. Since the damage of buffer overflow attacks is fatal and many of its products are based on C/C++, the company has to build its own buffer protection suites.

The first version of GS, buffer security check, came out with the release of visual studio 2002. The mechanism of GSv1 is similar to StackGuard, except that cookie is used instead of canaries.

At that time, the GS mechanism is simple and weak, although this mechanism is working with the exception mechanism. Local variables can be overwritten and the exception mechanism is easy to be bypassed.

The exception mechanism is to deal with unexpected situations. When those unintended situations occur, either the program catches them or the operating system handles them. The exception mechanism of Windows is based upon the Structured Exception Handling (SHE) mechanism. The SEH structure records the information of one program in the form of a chain, namely SEH chain. There are three key points of SEH: pointer to next SEH record, pointer to the exception handler, and end of the SEH chain.

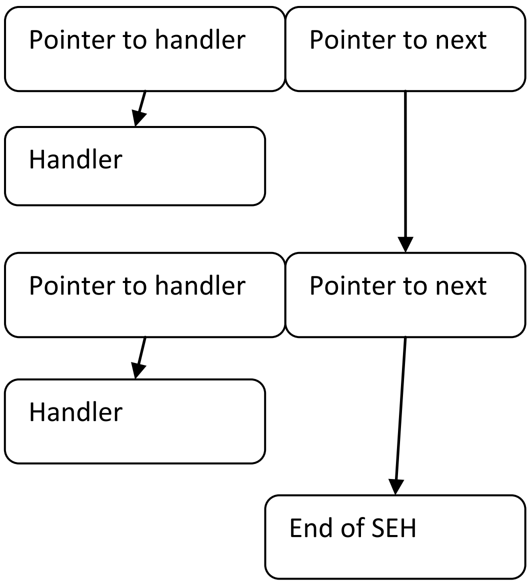


Figure 11 SEH chain

Attackers can get away with their malicious code by changing the exception mechanism. This is called the exception-handler hijacking. [21] By using this method, attackers can bypass the Microsoft Windows Structured Exception Handling mechanism. In Windows, the information of exception handling forms a linked list on the stack. With SEH records on the stack being overwritten, those pointers can be corrupted and set to point to attackers' shellcode, which might be a serious exploit.

Then, GSv1.1 released with visual studio 2003, in which SafeSEH is added. A function compiled with /GS returns and checks the canaries. The original canary is stored in data section and the other canary is on the stack. If the two matches, everything is fine and the program continues. Otherwise, a security handler should be revoked.[41]

But attackers can still use handlers on the stack outside the part which is compiled with SafeSEH. When the Windows XP SP2 came out, the road to exploits was much narrower than before. The OS is compiled with GS and SafeSEH, while DEP is introduced. So, there are hardware-enforced non-executable regions and software-enforced SHE validation.

If the base address of the stack, heap and libraries are randomly chosen upon startup of each process, or at some short time interval, the probability of a successful attack using return-into-libc is reduced. However, there are still ways to bypass NX mechanism, such as return-to-VirtualAlloc[42], disable-DEP via ProcessExecuteFlags[38], etc.

Development was made with Visual Studio 2005. GSv2 implements a shadow copy of parameters and employs strict GS pragma. When the Windows Vista was released, the ASLR technique was applied.

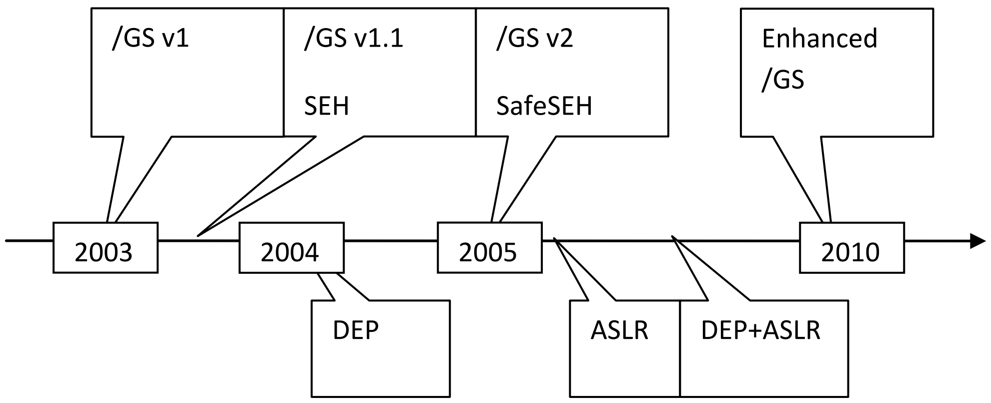


Figure 12 Timeline

At present, safe unlinking mechanism is applied in the new Windows 7 and new patches. The entropy for kernel level randomness is increased, since x64 OS has 8bits. With the release of Visual Studio 2010, Microsoft brings out the Enhanced GS. As the name indicates, this version of GS is enhanced. First of all, the GS heuristic is enhanced to take care of larger scope of variables than before. Another enhancement is to remove unnecessary GS cookies. [47]

### 3.1.2 Principles of /GS

No matter which version it is, GS is a compiler option, fundamentally using the same idea. The compiler heuristics find vulnerabilities, then GS inserts cookie into the stack frame in the prologue of the suspicious function, and finally the cookie is checked in the epilogue of the function [48]. By using some disassemble tools, we can find out GS tags in compiled programs. Moreover, the Microsoft provides a tool named BinScope, which is a binary analyzer to verify Microsoft's Security Development Lifecycle (SDL) requirements in developing software [49].

According to this design, firstly the GS do not protect all the functions. The GS mechanism needs to analysis the program and locates those at-risk functions. Secondly, it does not insert cookie for small size buffer. Thirdly, copies of vulnerable parameters will be made for recovery. The first and second points are designed for reducing overhead, while the third one is for safety.

Although the basic idea remains unchanged, enhancements and optimization are being made all the time. With the release of Visual Studio 2010, GS has got some enhancements, like heuristic-based approach to guess the safety of structures [50]. Since GS has got several versions with different overhead and security levels, different versions are applied to meet various requirements. This means that users can choose different level of GS protections according to the demands [46]. We can see from [47] that enhancements are made to widen coverage of potential vulnerabilities, while optimizations aim to remove unnecessary protections.

## 3.2 GS overhead

As mentioned above, the GS will not protect every single function. This reduces a lot of performance cost. However, in those protected functions, the overheads of GS protection can still cause performance loss.

The overhead of GS stack protection exists only in two places: the prologue and epilogue of the protected function. As described in the previous sections, the compiler will insert a small sized piece of data into the stack during the prologue of the function and check the data during the epilogue. The small piece of data added to the stack is generated randomly with the size being equal to the size of a pointer or an integer, and which is four bytes on win32 and eight bytes on x64. This piece of data named a security cookie is fairly small and is able to xor with the return address, since they are the same size.

To show the GS protection clearly, it is important to look at the assembly output of a piece of code. As the GS protection is a kind of compile-time optimization, the security cookie is added before the machine code is generated. Therefore, the way it works at the assembly level can be viewed.

The C code chosen contains only one function besides its main function. In the main function, the foo function is to be called and timed. This simply function will also be used to show the performance overhead of the GS stack protection. The code is like:

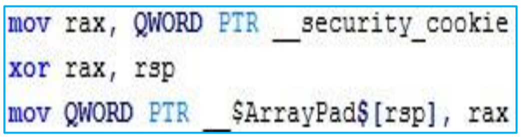
```
void foo(){

        int buffer[20];

        buffer[0]=1;

}
```

The source code was compiled with the command line like:

*cl code.c /Od /GS /FAs*

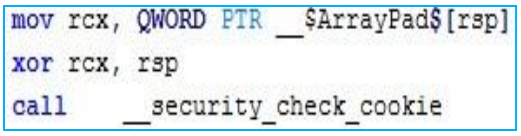
The cl is the Visual Studio compiler. “/Od /GS /FAs” are option flags. “/Od” disables optimizations, such as inline function expansion, intrinsic functions, favour size or speed, omit frame pointers, fiber-safe, and whole program optimization [51]. “/FAs” tells the compiler to output the assembly code. “/GS” is the GS protection option flag. “/GS-” will disable the GS protection.

The main difference between the assembly code files generated from the same source code with “/GS” and “/GS-” is obvious through the comparison. The GS protected version has more lines in the prologue and epilogue parts of the function. On the x64 platform, the extra lines are shown as follows:



```
mov rax, QWORD PTR __security_cookie
xor rax, rsp
mov QWORD PTR __$ArrayPad$[rsp], rax
```

Figure 13 Prologue assembly code

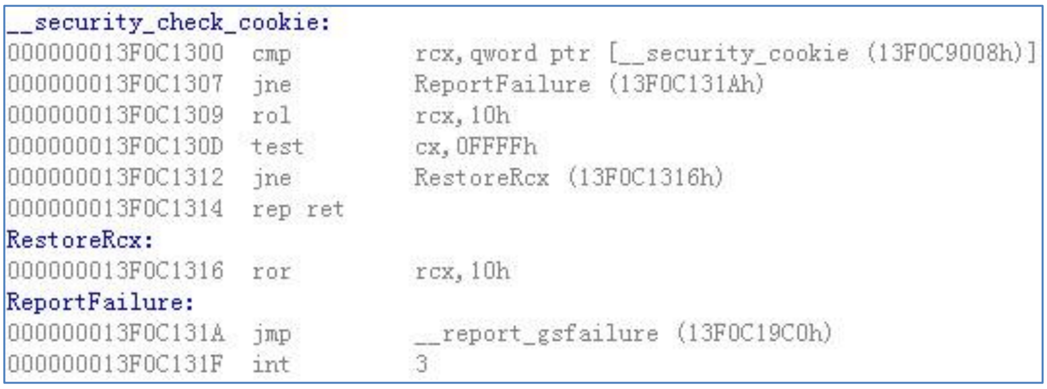


```
mov rcx, QWORD PTR __$ArrayPad$[rsp]
xor rcx, rsp
call     __security_check_cookie
```

Figure 14 Epilogue assembly code

During the prologue, the __security_cookie acts as a pointer pointing to a globe variable. This globe variable is the random data mentioned above. This small piece of data is first put in the rax register and then completes a XOR operation with rsp, stack pointer pointing to the top of stack. The result of the XOR operation will be pushed into the stack as the ArrayPad.

During the epilogue, the ArrayPad performs a XOR operation with rsp, which is supposed to recover the __security_cookie. Then, the __security_check_cookie function is called upon in order to check whether the __security_cookie is corrupt.

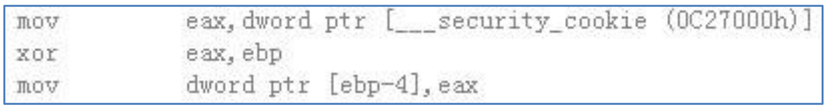


```
__security_check_cookie:
000000013F0C1300  cmp        rcx,qword ptr [__security_cookie (13F0C9008h)]
000000013F0C1307  jne        ReportFailure (13F0C131Ah)
000000013F0C1309  rol        rcx,10h
000000013F0C130D  test       cx,0FFFFh
000000013F0C1312  jne        RestoreRcx (13F0C1316h)
000000013F0C1314  rep ret
RestoreRcx:
000000013F0C1316  ror        rcx,10h
ReportFailure:
000000013F0C131A  jmp        __report_gsfailure (13F0C19C0h)
000000013F0C131F  int        3
```

Figure 15 security check function

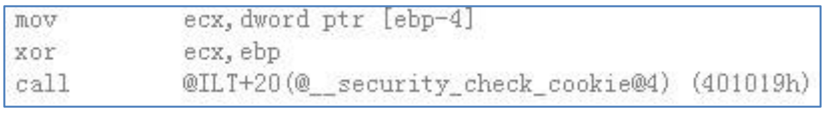
As shown above, the __security_check_cookie function compares the result of XOR with the __security_cookie. If the two values are not equal, the program will report a failure.

On the win32 platform, the procedure is mostly the same, except that it is ebp registered in the __security_cookie do XOR with.



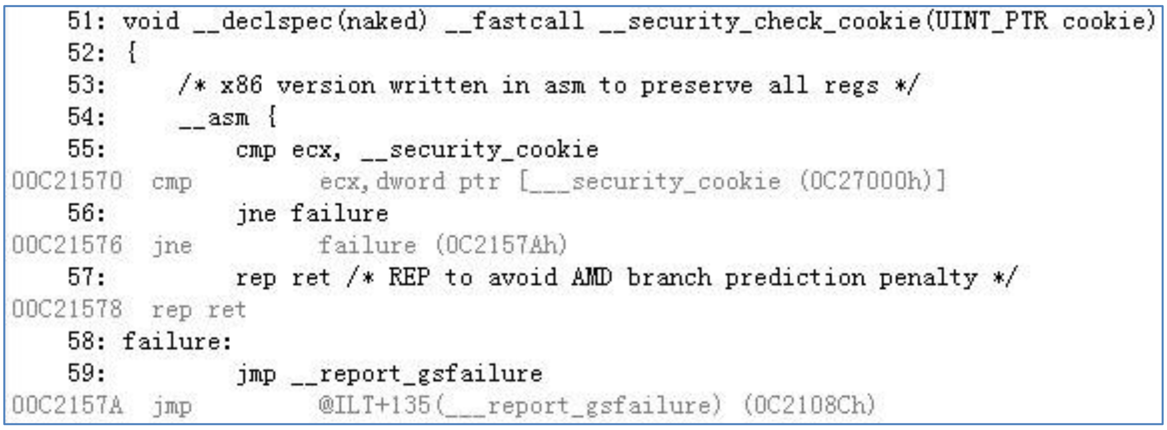
```
mov         eax,dword ptr [___security_cookie (0C27000h)]
xor         eax,ebp
mov         dword ptr [ebp-4],eax
```

Figure 16 Prologue on Win32



```
mov         ecx,dword ptr [ebp-4]
xor         ecx,ebp
call        @ILT+20(@__security_check_cookie@4) (401019h)
```

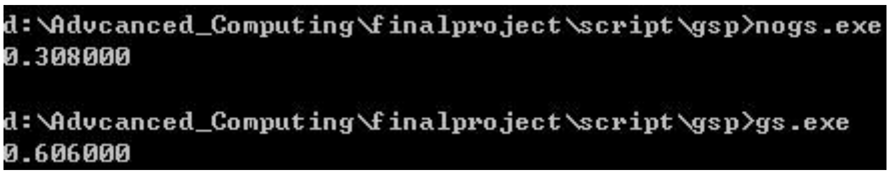
Figure 17 Epilogue on Win32



```
    void __declspec(naked) __fastcall __security_check_cookie(UINT_PTR cookie)
    {
        /* x86 version written in asm to preserve all regs */
        __asm {
            cmp ecx, __security_cookie
00C21570  cmp         ecx,dword ptr [___security_cookie (0C27000h)]
            jne failure
00C21576  jne         failure (0C2157Ah)
            rep ret /* REP to avoid AMD branch prediction penalty */
00C21578  rep ret
    failure:
            jmp __report_gsfailure
00C2157A  jmp         @ILT+135(___report_gsfailure) (0C2108Ch)
```

Figure 18 Security check function on Win32

In the environment of 64-bit, the source code was compiled, outputting nogs.exe without GS protection and gs.exe with GS protection. The two programs can print their own execution time in second on the terminal.



```
d:\Advcanced_Computing\finalproject\script\gsp>nogs.exe
0.308000

d:\Advcanced_Computing\finalproject\script\gsp>gs.exe
0.606000
```

Figure 19 Execution time (64-bit)

The program without GS protection runs obviously faster than the one with GS protection. But the execution time of gs.exe is only twice that of the execution of nogs.exe, which means that the extra operations of GS protection have the same time overhead as the operations in the function. Thus, at least in this case, the cost of GS protection is no more than one assignment operation of an array.

For x86 32-bit, similar commands were used to compile the same source code, outputting n.exe without GS protection and y.exe with GS protection.

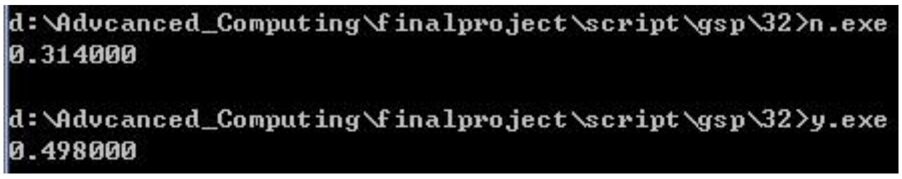

```
d:\Advcanced_Computing\finalproject\script\gsp\32>n.exe
0.314000

d:\Advcanced_Computing\finalproject\script\gsp\32>y.exe
0.498000
```

Figure 20 Execution time (32-bit)

As the result shows, the cost of GS protection is less than one assignment operation of an array.

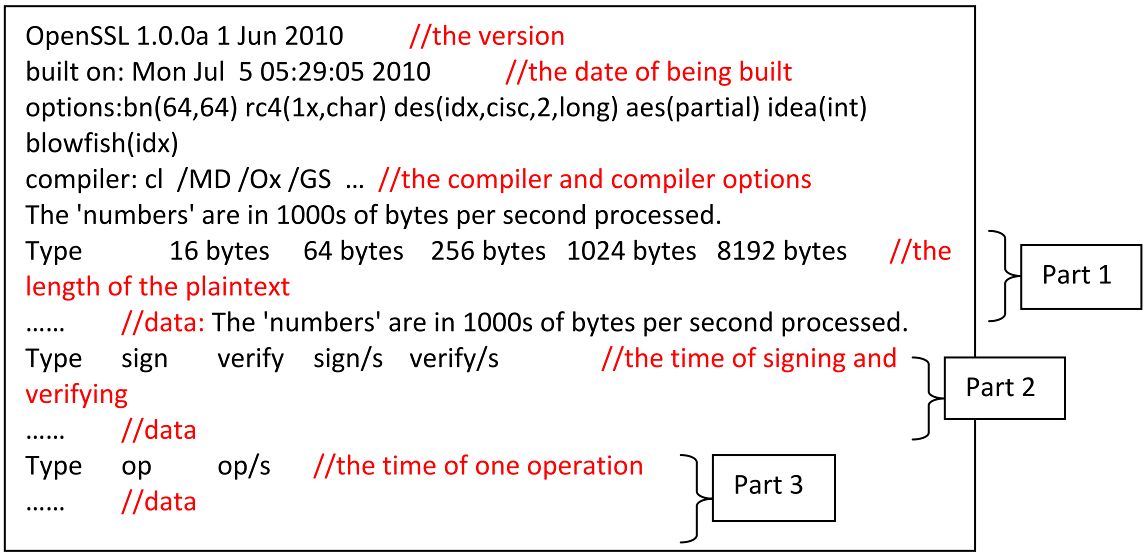
Overall, the overhead of GS protection is relatively quite small.

**3.3 OpenSSL speed test**

According to Tim Burrell from Microsoft Security Engineering Center (MSEC) and my supervisor Daniel Page, there are some cases showing that the performance of GS protected program may be improved somehow. To obtain evidence to support that phenomenon, the OpenSSL was chosen to be compiled with GS and without GS option to test the performance of multiple functions with various algorithms.

OpenSSL is a toolkit with the implementation of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols [52]. There are two reasons for us to choose OpenSSL. Firstly, it is an open source toolkit and we can access its source code freely. In this case, it can be compiled through the author's own preference. It is also helpful to look at the source code while studying the performance of the program. Secondly, the OpenSSL toolkit contains an excellent cryptographic library. With its cryptographic library, we can test a large number of cryptographic algorithms. Since the OpenSSL provides a coherent framework and user-friendly interface, this library is easy to use.

As for testing, the OpenSSL has a small tool named "speed" to test the algorithms in the cryptographic library. It is invoked when we run OpenSSL with an option "speed" in command line. The synopsis of the speed option is shown on [53]. With the command "OpenSSL speed", the OpenSSL version 1.0.0a will test the performance of all the algorithms in the library. For chosen algorithms, the option after the word "speed" can be also specified. The output of the above commands can be redirected into a file, in which the data format may look like this:



```
OpenSSL 1.0.0a 1 Jun 2010        //the version
built on: Mon Jul  5 05:29:05 2010        //the date of being built
options:bn(64,64) rc4(1x,char) des(idx,cisc,2,long) aes(partial) idea(int)
blowfish(idx)
compiler: cl  /MD /Ox /GS  …  //the compiler and compiler options
The 'numbers' are in 1000s of bytes per second processed.
Type          16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes      //the
length of the plaintext
……       //data: The 'numbers' are in 1000s of bytes per second processed.
Type     sign       verify    sign/s    verify/s           //the time of signing and
verifying
……       //data
Type     op         op/s      //the time of one operation
……       //data
```

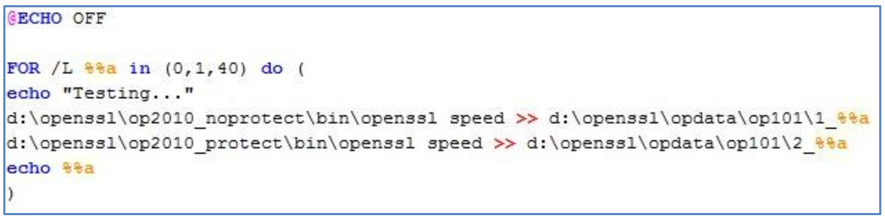
Figure 21 Output of OpenSSL speed

In the above picture, the data is categorized into three parts while there are also C-style single-line comments beginning with two forward slashes. These comments explain what those lines are. Knowing the format of the output can help process the data. In this analysis, only the part 1 of the output data was used to show the result.

At this stage, we can go on talking about the testing of OpenSSL speed.

The first step in testing OpenSSL speed performance is to compile the OpenSSL source code through the Visual Studio compile environment with /GS[-]. Under the root directory of OpenSSL source code, files prefixed with "INSTALL" describe how it is compiled on several different platforms. According to those instructions, in the environment of Visual Studio /GS or /GS- flag to the "ntdll.mak" file under the "ms" directory can be added.

Then, as part of the second step, some batch script files are written to let the compiled OpenSSL programs run many times and redirect their output to separate files. The batch script looks like:



```
@ECHO OFF

FOR /L %%a in (0,1,40) do (
echo "Testing..."
d:\openssl\op2010_noprotect\bin\openssl speed >> d:\openssl\opdata\op101\1_%%a
d:\openssl\op2010_protect\bin\openssl speed >> d:\openssl\opdata\op101\2_%%a
echo %%a
)
```

Figure 22 Batch script for testing

Finally, the third step, the data files from the second step are collected and processed, outputting a file containing the data in the same format of the original files. Specifically speaking, the original files are treated as matrix and the average of data from the same OpenSSL compiling versions (GS protected or not) is gained. Then the difference of the two averaged data is calculated and divided with the average of the no-GS protection version's data. This step is done using Python.

The following graph shows the workflow of the OpenSSL testing described above.

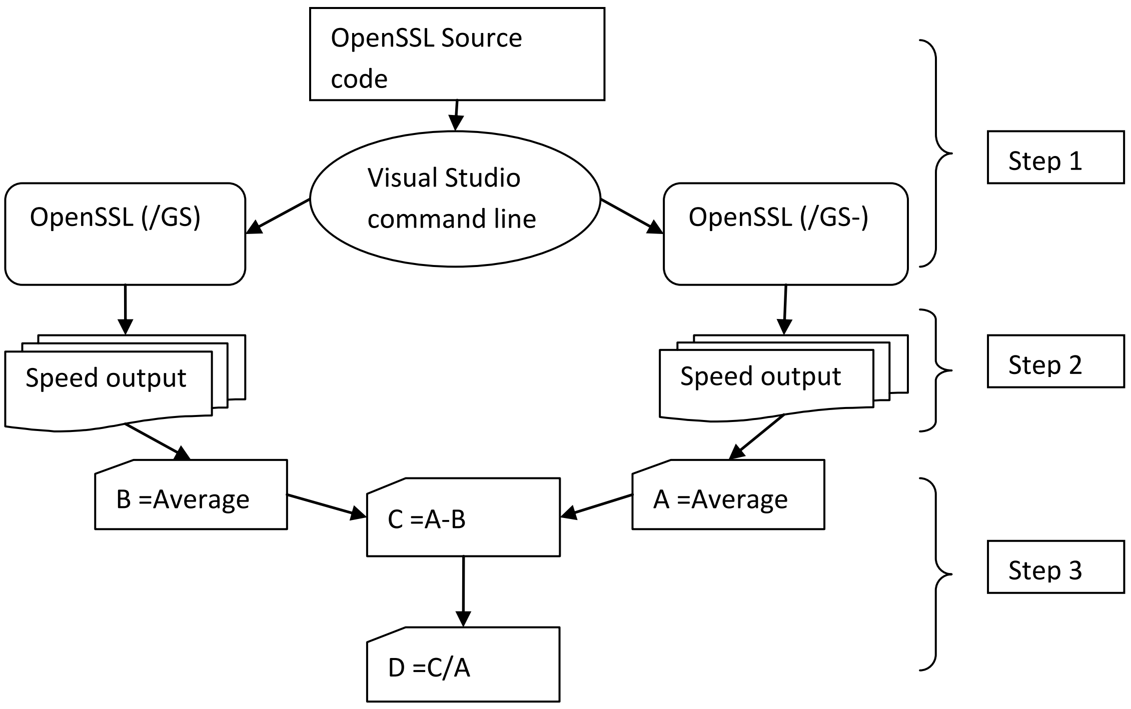


Figure 23 Workflow of the testing

The operations on data are similar to matrix operations (matrix subtraction, matrix division). At the end of the workflow, a file named D containing numbers was produced.

Since in the output format there are three part containing different data types, the data in D is not coherent. Therefore, these have to be investigated separately.

a) For the first part of the D, the data is the ratio of the per-second-processed operation difference to the per-second-processed operation of the no-GS version. This indicates that the GS version has a better performance than the no-GS version if the number is positive. Otherwise, the no-GS version has a better performance.

b) In the second part, only the columns for signing time and verifying time are processed into D. The data in this part of D is the ratio of the signing/verifying time difference to the signing/verifying time of the no-GS version. This indicates that the GS version has a better performance than the no-GS version if the number is negative. Otherwise, the no-GS version has a better performance.

c) The third part is similar to the second. Only the column of op is processed into D. The data in this part is the ratio of the operation time difference to the operation time of the no-GS version. This indicates that the GS version performs better than the no-GS version if the number is negative. Otherwise, the no-GS version has a better performance.

With Visual Studio 2010 x32, 124 output files were examined, including 62 files generated from the GS protected version and 62 from the no GS protection version. From this, there materialises a huge set of data, because “OpenSSL speed” tested about 67 ciphers. With the help of Excel, the following graph shows how the performances of the GS and no-GS versions differ.

For the first part of the speed output:

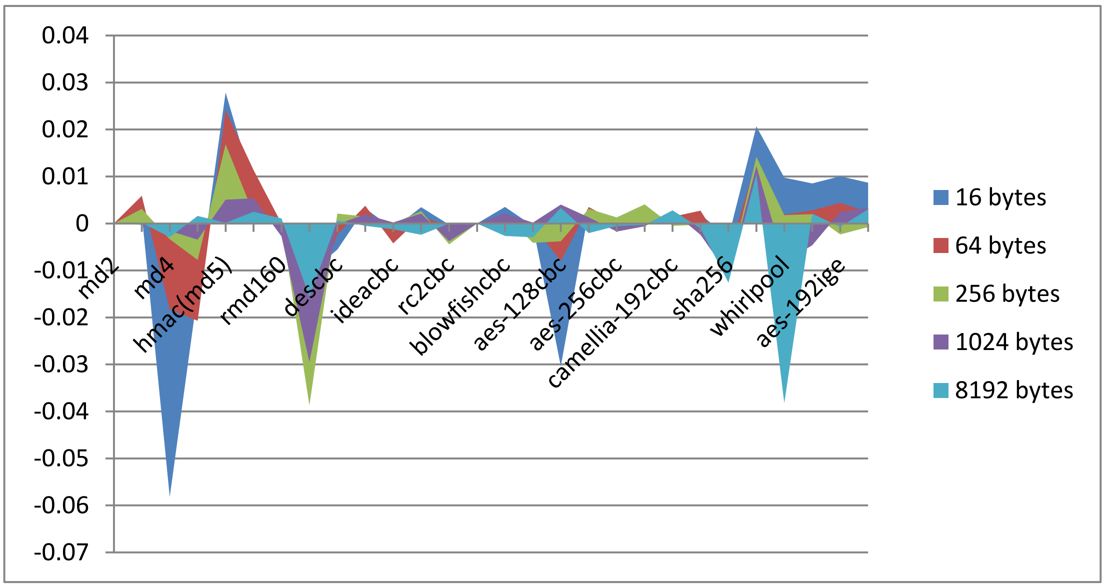


Figure 24 Performance of various algorithms in OpenSSL

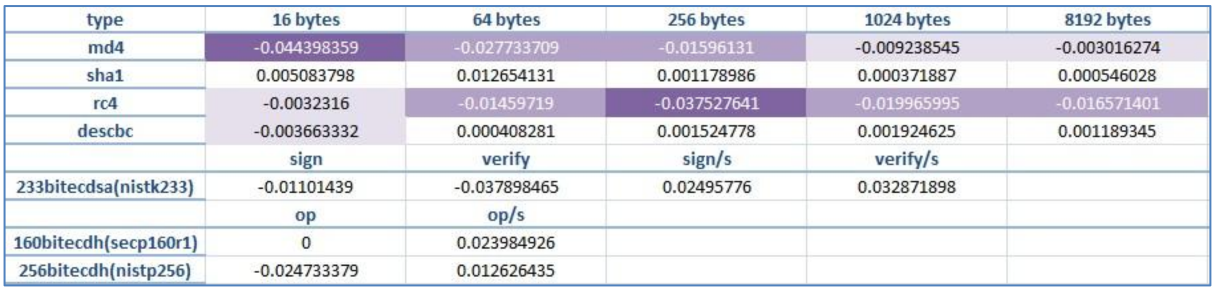
The horizontal axis lists a number of cryptographic algorithms and the vertical axis is the data of D described above. Hence, this graph shows that algorithms like MD4 are obviously faster with GS than without GS.

## 3.4 Analysis and further testing

The outcome results in the above parts showing that the programs with certain algorithms can run faster with GS stack protection than without it.

One round of testing may not be enough. In order to be more convinced about the counter-intuitive phenomenon, further testing had to be done. Ciphers, which have obviously perform better with GS protection in previous testing, were chosen to be tested again with OpenSSL speed command.

Then, the OpenSSL speed command with selected ciphers was executed 702 times, generating 702 files. The result is shown as follows:



| type | 16 bytes | 64 bytes | 256 bytes | 1024 bytes | 8192 bytes |
|---|---|---|---|---|---|
| md4 | -0.044398359 | -0.027733709 | -0.01596131 | -0.009238545 | -0.003016274 |
| sha1 | 0.005083798 | 0.012654131 | 0.001178986 | 0.000371887 | 0.000546028 |
| rc4 | -0.0032316 | -0.01459719 | -0.037527641 | -0.019965995 | -0.016571401 |
| descbc | -0.003663332 | 0.000408281 | 0.001524778 | 0.001924625 | 0.001189345 |
|  | sign | verify | sign/s | verify/s |  |
| 233bitecdsa(nistk233) | -0.01101439 | -0.037898465 | 0.02495776 | 0.032871898 |  |
|  | op | op/s |  |  |  |
| 160bitecdh(secp160r1) | 0 | 0.023984926 |  |  |  |
| 256bitecdh(nistp256) | -0.024733379 | 0.012626435 |  |  |  |

Figure 25 Performance of Selected algorithms

The data are calculated through the same procedure as the previous. The numbers in the first five rows are from (nogs – gs)/nogs, in which, nogs represents the number of operations per 1k seconds that the no-GS programs performed, while gs is the number of operations per 1k seconds that the GS protected programs performed, If the number is positive, it means the GS protected program performed less operations per 1k seconds than the program without GS protection. If the number is negative, it means that the GS protected program carried out more operations. Thus, the above table shows that the MD4 and RC4 algorithms have better performance with GS protection than without GS protection.

We also looked at the stack protection techniques in GCC, named Stack-Smashing Protector (SSP) [54]. The testing procedure is the same, while the environment is set to GCC compiler on the Linux platform. After having compiled the OpenSSL using GCC options –fstack-protector and –fno-stack-protector separately, the programs were executed 40 times. The result for the first part of the output is:

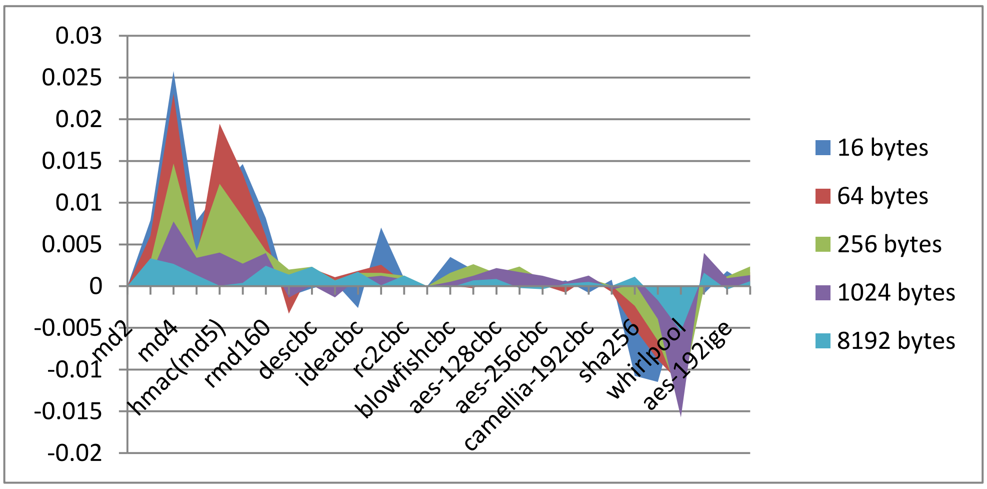


Figure 26 Result of GCC SSP

In the graph, lines going below 0 indicate that the algorithms had better performance with SSP protection than without it. Therefore, there are several algorithms showing that phenomenon, such as whirlpool. This brief experiment indicates that the SSP protection can also boost the performance of programs. Since the basic idea of SSP is similar to the GS stack protection's mechanism, the performance boost may not only occur within the GS protection.

In conclusion, the OpenSSL was used to show some strong evidence supporting the idea that the GS protection can somehow improve the performance of certain programs. This may appear unusual and atypical to common sense. Then, this counter-intuitive phenomenon was studied to understand the reasons for this, which may shed light on the optimization of the compiler.

# Chapter 4. Hypothesises and Verifying

## 4.1 Methodology

Chapter 3 shows that cryptographic algorithms in OpenSSL cryptographic library like MD4 have a better performance when compiled with /GS than without /GS-. Thus, the evidence appears to be reasonably concrete that the GS protection can somehow boost the performance of certain programs. Accordingly, one of our aims was to figure out the possible reasons that the stack protection improves the performance of certain programs. That formed the basis for the study.

It may be a normal thinking to do further research by investigating and debugging the source code of those algorithms which show better performance in the previous testing experiments. As the GS protection is added in compile-time, we can look at the assembly language versions of the programs.

However, it proved very difficult to find clues with ease by looking at the source code, or even the assembly. Needless to say, the assembly code from exactly the same source code is different, only having some extra code for the security cookie and unequal data for the stack.

As a normal consideration, one or two algorithms were chosen as they supported the phenomenon in the previous experiments. One was MD4, the other is DES-CBC. However, problems appeared: First, the algorithms were based on other components of the OpenSSL. For example, the core of this algorithm under the directory of crypto/md4 includes several other files whilst being compiled. It would be a huge task to debug one algorithm. Second, there are some codes generated from Perl files in the OpenSSL. This makes things more complex.

Generally speaking, it may not be smart to look into the source files of the OpenSSL. After investigating some of the source code, no clues could be found to explain the phenomenon.

According to above analysis, it may not be a good way to look at the algorithms. Looking at the issue from a different perspective may help: the aim is to figure out why and how the GS stack protection mechanism has the ability to improve performance in some situations, instead of finding out why the algorithms, say MD4, run faster with GS option on. This indicates that there should be some theories supporting the phenomenon.

Thanks to my supervisor Daniel Page, I have found the right way to do the research. The methodology is provided in order to think of possible theoretical reasons and verify them through experiments. This method widens the area of the research, so it is not limited to studying algorithms in the OpenSSL.

Therefore, in the following parts of this chapter, several hypotheses will be formulated which will be verified in next chapter.

## 4.2 Hypothesis 1: Memory paging influences the performance

### 4.2.1 Basic idea

The first hypothesis is that the performance improvement with GS protection is caused by memory paging.

As a vital layer between hardware systems and software applications, the operating systems are responsible for memory management. The upper layer of software applications is normally innocent of the underneath hardware like CPUs, RAMs and hard disks, which means that the programmers cannot directly change the electric potential on the physical RAM. When the programmers manipulate the data on the memory, they are actually accessing a virtual memory that is mapping the physical one.

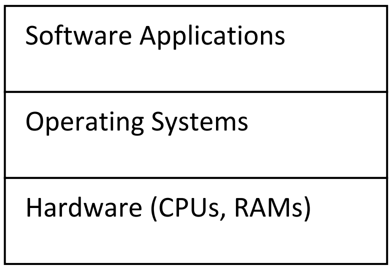


Figure 27 Layers of the computer architecture

It is complex and not easy for programmers to check the addresses on the physical main memory, RAM. The RAM is normally expensive and small compared to hard disk, but the various applications demand huge size of memory for each of them.

On the Windows system, Microsoft provides Windows APIs for upper layer applications to manage memory without actually access the physical hardware. Here is a picture taken from MSDN showing the memory management in Win32.

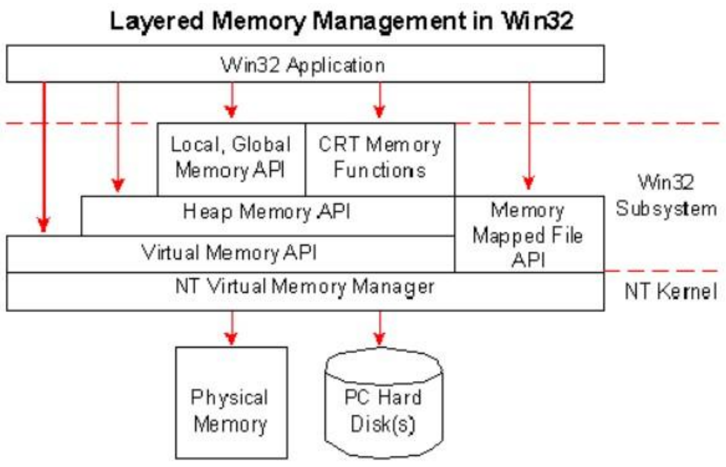


Figure 28 Layered Memory Management in Win32 Taken from [55]

Hence, what the programmers are managing is actually the virtual memory. According to [56], the virtual memory space is set for each program/process in order to distinguish its memory space from others. Programs operate the virtual addresses on the virtual memory space and those virtual addresses are to be translated to the actual addresses on the hardware by a memory management unit (MMU). But the virtual memory space is not implemented as a whole space. It is normally spitted into pages. This memory management scheme is called paging [57]. The default page size is 4k bytes (4096 bytes) on most platforms.

According to the document[58] and [59] on Microsoft's website, the Visual Studio compiler inserts a function named __chkstk() during the prologue of a function when the compiler has to allocate larger space than one page.

This is a nice aspect to look at for potential reasons of the performance boost with GS protection. For one, at the boundary of memory pages, the performance of programs may be unpredictable. Plus, the helping function __chkstk might have something to do with the stack protection mechanism. In the following part, we will discuss those issues more deeply.

### 4.2.2 Experimental Validations

To verify the first hypothesis, the stack of size passing the page size should be tested. In order to do so, we'd better create functions with stack of continuous increasing sizes. Through testing the functions with and without GS protection, this hypothesis can be verified.

#### 4.2.2.1 Experiment Design

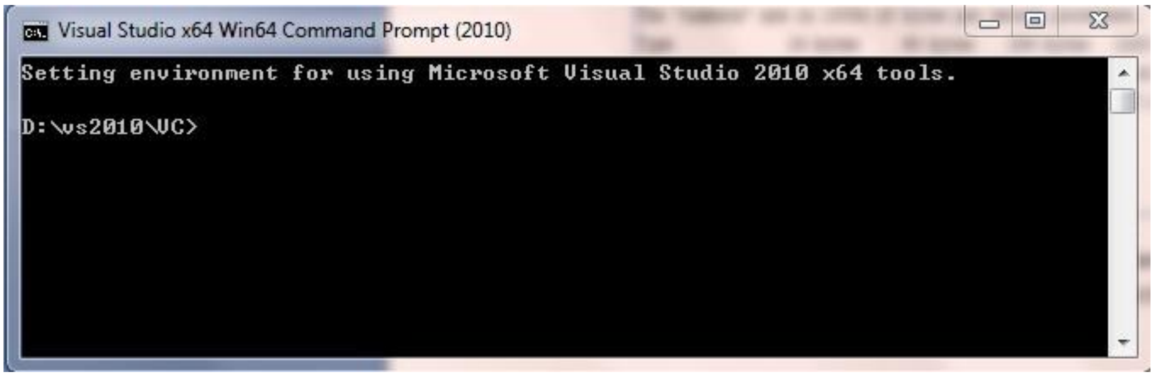
The procedure of our experiment is like this:

a. First, suitable functions are designed. We use the main function to call a function and time the execution time of the callee function. The pseudo code of main function is like this:

```
int main(){
    Local variables;
    Begin the timing;
    For-loop 100000 times {
        Call the function foo();}
    End the timing;
    Output the execution time;
    return;
}
```

The foo function is what we are going to think of. This foo function must contain at least one GS buffer [50].

b. Then, those functions were compiled with and without GS protection. This might be done by command like: cl /GS[-] /Od foo.c. In our project, we mainly focus on the 64-bit system, so the compiling is done on Visual Studio x64 Command Prompt.



```
Setting environment for using Microsoft Visual Studio 2010 x64 tools.

D:\vs2010\VC>
```

Figure 29 A snapshot of Visual Studio 2010 x64 Command Prompt

c. After that, the outcome programs were executed and the execution times were collected. Batch script was used to implement this step, redirecting outputs into text files.

d. Last, the data in the files is to be copied into Excel and graphs are generated for further analyzing.

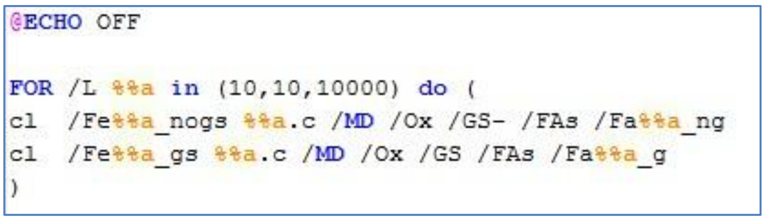
#### 4.2.2.2 Experiment: String copy

To study the behaviour of the program whose stack is across pages, functions with continuous stack sizes shall be generated. The simplest kind is the function that can copy a string to a local buffer.

```
void foo(char* ba){

        char buffer[10];

        strcpy(buffer,ba);

}
```

So, we created thousands of C files containing that kind of foo function. In those source files, the size of buffer goes from 10 to 10000, increasing by 10. The argument “ba” is a char array defined in the main function as a string of the same size of buffer.  Some small pieces of Python were written to generate those files.

After being generated, the source files were compiled by a small piece of batch script:



```
@ECHO OFF

FOR /L %%a in (10,10,10000) do (
cl  /Fe%%a_nogs %%a.c /MD /Ox /GS- /FAs /Fa%%a_ng
cl  /Fe%%a_gs %%a.c /MD /Ox /GS /FAs /Fa%%a_g
)
```

Figure 30 Batch script of compiling the source files

To run the outcome programs, another piece of batch script was written in the same way, except that the outputs were redirected to a text file.

Then these data was collected in Excel and a graph was generated as below. In the graph, the vertical axis displays the execution time and the horizontal axis shows the size of the buffer in the function. The blue line is the data from no-GS protection version of programs, while the red line is the data from programs with GS protection. The green line is the absolute difference of execution time. (The reasons of using the absolute difference instead of relative difference are: the execution times of both versions are already on the graph, and the absolute difference of execution time is enough for showing the behaviour.)

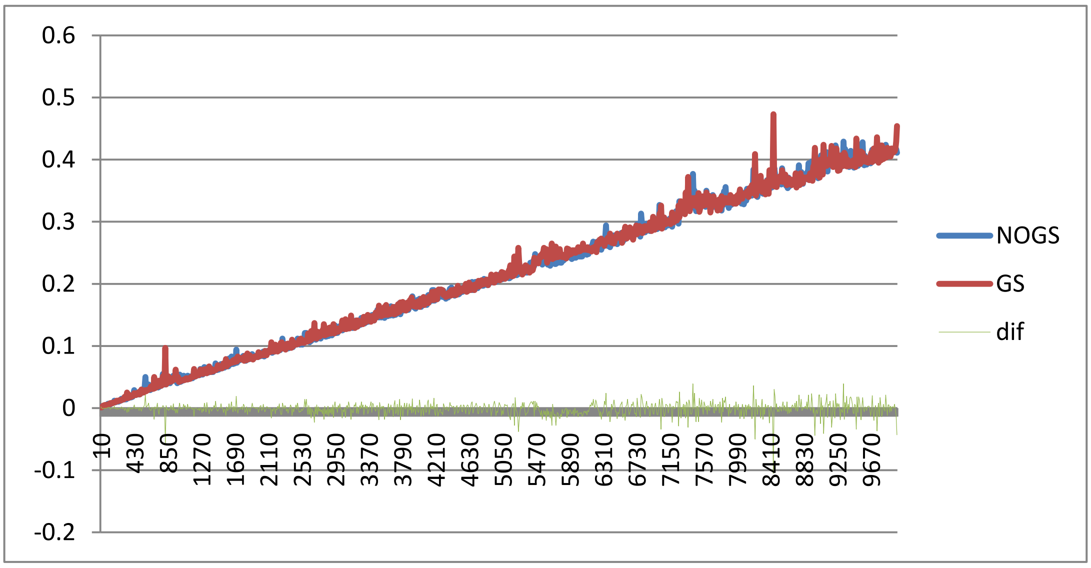


Figure 31 Data of Experiment 1

According to this graph, the difference is minor between two versions as the green line is fluctuating slightly around 0. This is because that the performance loss on GS protection is quite small and there is little performance boost with GS protection.

Looking back at the compiling options, we can find that the /Ox option may optimize the source code and cause some undefined effect to our experiment. Moreover, calling the strcpy function might affect the performance, too. So, in our further experiments, interference factors should be eliminated as many as possible.

Therefore, we continued the research with compiler option /Od (disable other optimization in Visual Studio) and using other adjusted functions.

#### 4.2.2.3 Experiment: Access only two pieces of data

Since the stack is across pages, it might show some differences between when the two pieces of data are in the same page and when they are not. Furthermore, with the existence of GS protection cookies, two pieces of data on the same page may be shifted to two different pages. To test this situation, we came up with the following function:

```
void foo(){
        int buffer[N];
        buffer[N-1]=10;
        buffer[N-500]=20;
        buffer[N-500]+=buffer[N-1];
}
```

In this function, two integers of the buffer array are accessed. With the size of the buffer, N, increased from 7200 to 8999, the stack frame of this function should exceed the page size 4k bytes.

The result of this function is as follows:

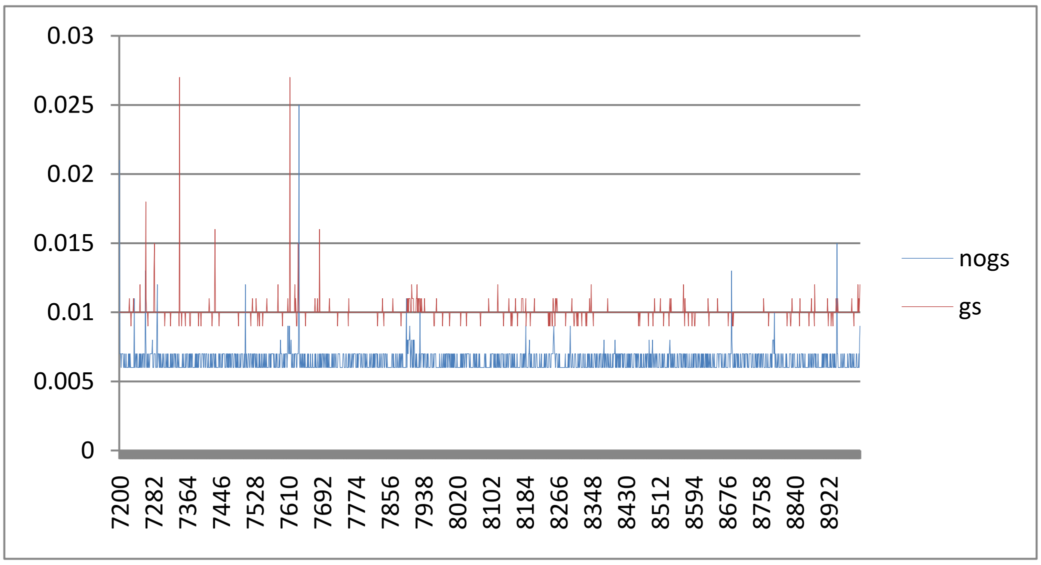


Figure 32 Data of Experiment 2

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection.

According to the graph, in average, the execution of GS version takes 0.5 times longer than the one without GS protection. Since there are three assignment operations in the function, operations of the GS protection take the same time of one assignment operation, just like what we discussed in previous chapter. So, even if there is some performance boost, say 5% in one operation, it will not be visible because of the small amount of total operations. The point is that **the performance boost is too small to even cover the performance loss of GS protection.**

#### 4.2.2.4 Experiments: Go through an array to assign values

To make the potential performance gain more obvious, we can access more data on the stack. Then, the function as follows is designed:

```
void foo(){
        int buffer[N];
        int j;
        for(j=0;j<N;j++){
                buffer[j]=1;}
}
```

In the above pseudo code, N is the size of the integer array "buffer". About 1800 source code files were generated through a small piece of Python code with constant N from 7200 to 9000. With each N, the array has to perform assignment operation N times.

Then the result is:

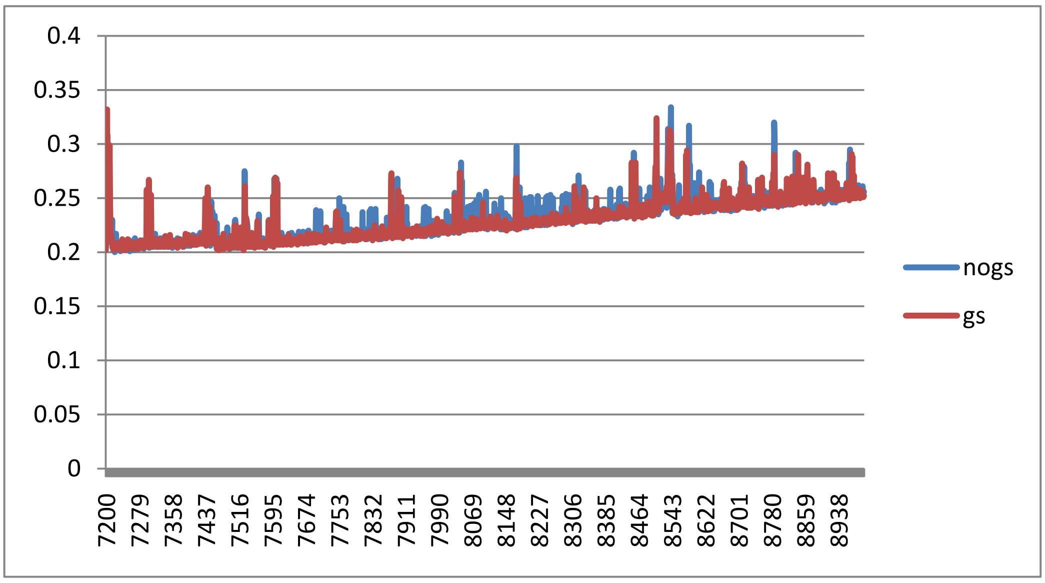


Figure 33 Data of Experiment 3

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection.

This graph shows that sometimes the performance really gets obviously boosted when N is around from 8000 to 8300.

That is a positive result. It shows that this kind of functions can has better performance with GS protection than without it, if the array size is suitable.

So, we go on with this function. This time the size of the buffer N changes from 512 to 3499. The result is:

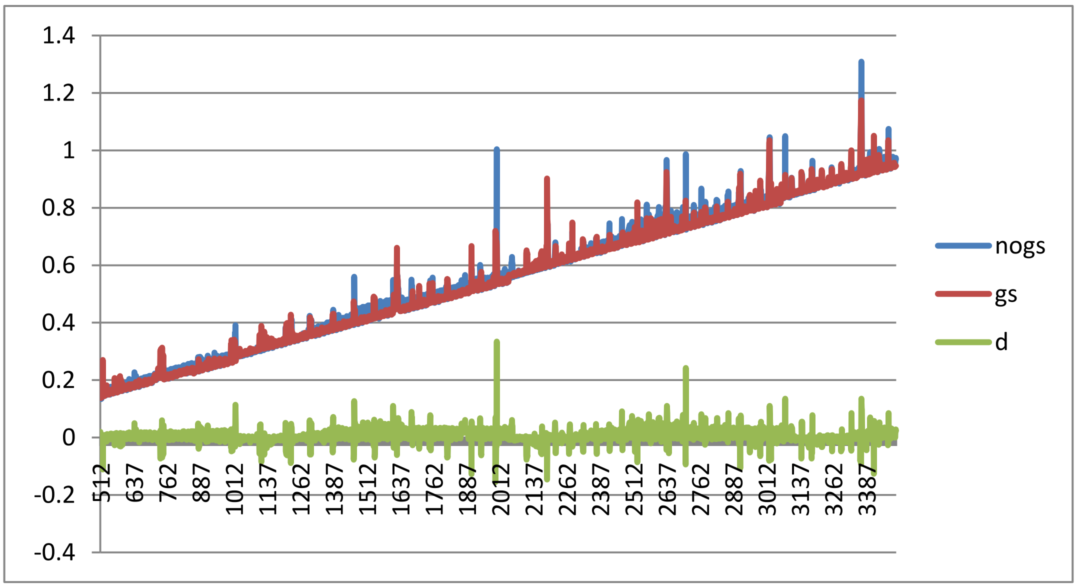


Figure 34 Data of Experiment 4

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. The green line is the absolute difference of execution time (d=nogs-gs).

The green line in the graph hovers above the 0 horizontal line, which means the programs with GS option on run faster than those without GS protection in general. Moreover, looking at the shape of the d, we can see the regular undulating. If the period between two down points is a round, there are about four rounds in this graph. Only two rounds are completed: from about N=1087 to N=2122, and from about N=2122 to N=3157. These numbers are according to graph and not so precise, but the length of the round's span is certainly about 1024. Since the size of int type is 4 bytes, the span of the fluctuating cycle is about 4K bytes, which **matches the size of one memory page**.

To ensure that it is not coincidence, I have compiled and run the programs several more times, and the results are the same. However, we'd better look into the fluctuating cycle to see if we can find more details.

Therefore, a part of the previous experiment, with N from 512 to 1450 are selected and tested one more time.

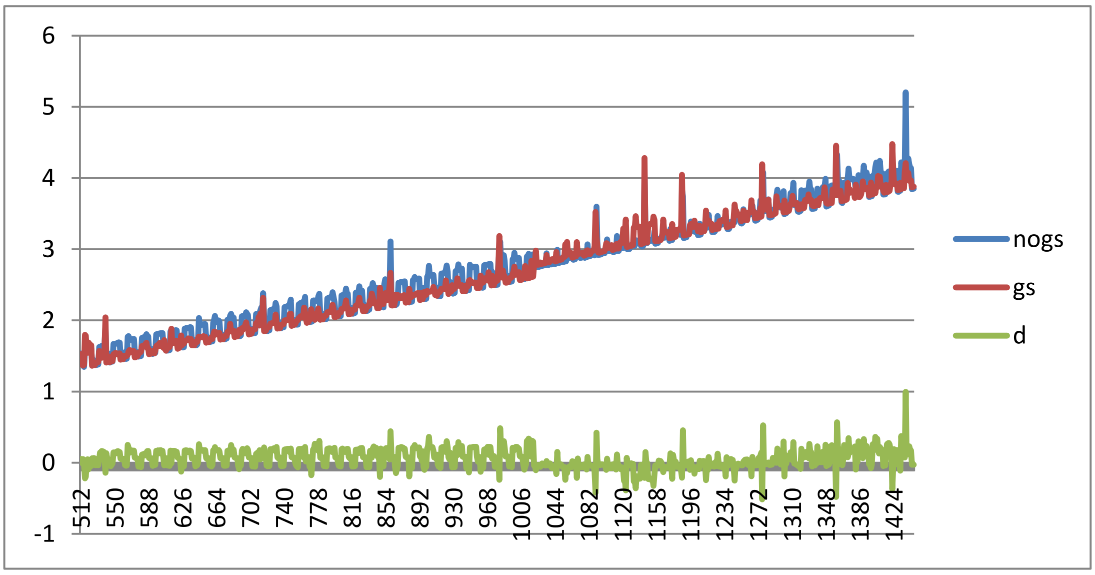


Figure 35 Data of Experiment 5

In this line chart, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. The green line is the absolute difference of execution time (d=nogs-gs).

According to the red and blue lines in the above chart, the lines shift upward after some point around N=1016. At that point, the programs are probably crossing the two pages as the stack size exceeds the limit size of one memory page. Then a helper function named __chkstk is called. This will be discussed in later part.

In our experiment, we found that some variants of the previous function show the same performance.

For example, this one:

```
int foo(){
        int buffer[N+1];
        int j;
        for(j=0;j<N+1;j++){
```

*buffer[j]=1;}*

*buffer[N]+=buffer[N-499];*

*return buffer[N];}*

The N is a constant integer, which varies from one file to another.

The result is:

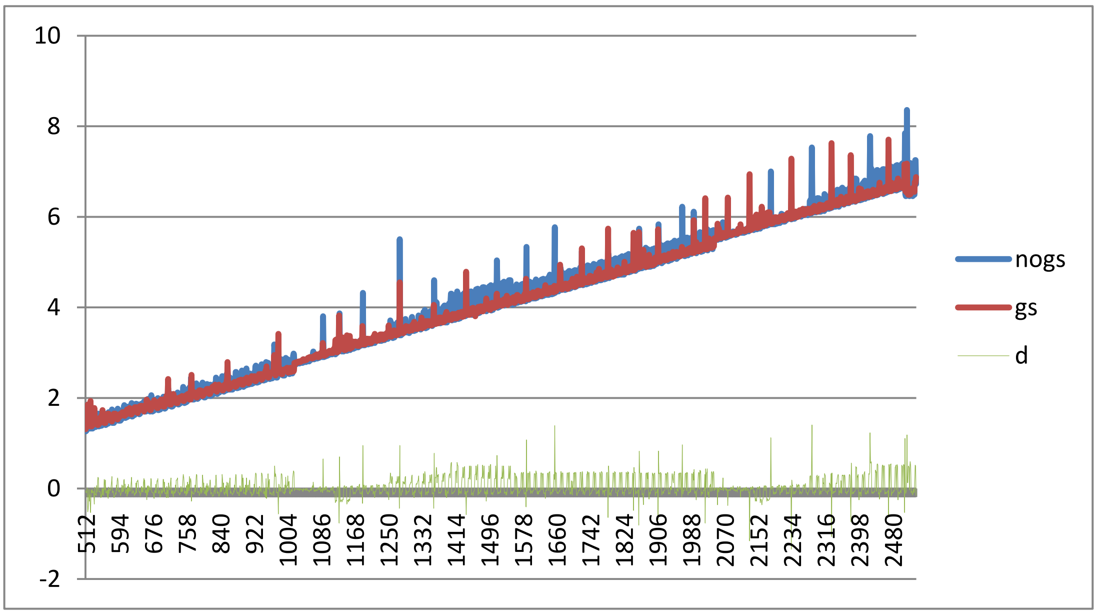


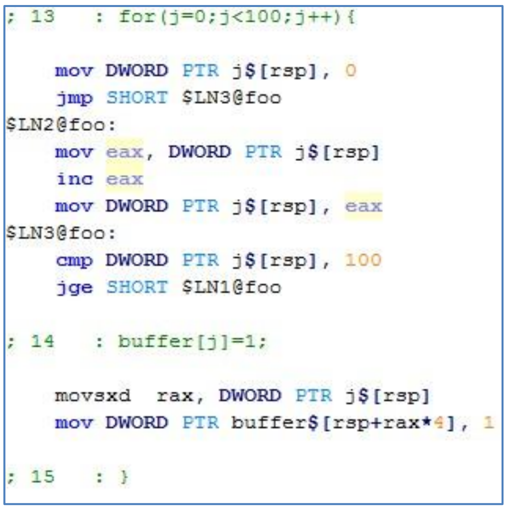
Figure 36 Data of Experiment 6

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. The green line is the absolute difference of execution time (d=nogs-gs).

This is evidence supporting the previous results. There is one complete round of performance difference in the chart. The fluctuation cycle of D is obvious and matches the size of a memory page. Moreover, at the beginning and end of one round of d, the performance of programs with and without GS protection also gets shifted upwards (at about N=1000 and 2024).

#### 4.2.2.5 Experiment: Rule out the for-loop interference

Still, there is one question: what influence might the for-loop in our functions have to the performance? The straight forward way is to look at the assembly version of the code.



```
; 13    : for(j=0;j<100;j++){

    mov DWORD PTR j$[rsp], 0
    jmp SHORT $LN3@foo
$LN2@foo:
    mov eax, DWORD PTR j$[rsp]
    inc eax
    mov DWORD PTR j$[rsp], eax
$LN3@foo:
    cmp DWORD PTR j$[rsp], 100
    jge SHORT $LN1@foo

; 14    : buffer[j]=1;

    movsxd  rax, DWORD PTR j$[rsp]
    mov DWORD PTR buffer$[rsp+rax*4], 1

; 15    : }
```

Figure 37 For-loop assembly code

Above is a piece of code with its assembly code. Without any optimization, this piece of code will let the program access the stack 100 times to assign 1 to an int type data on the buffer.

In case the for-loop produces noise data, we let the for-loop be empty to check if the for-loop might contribute interference to the behaviour above.

The result is:

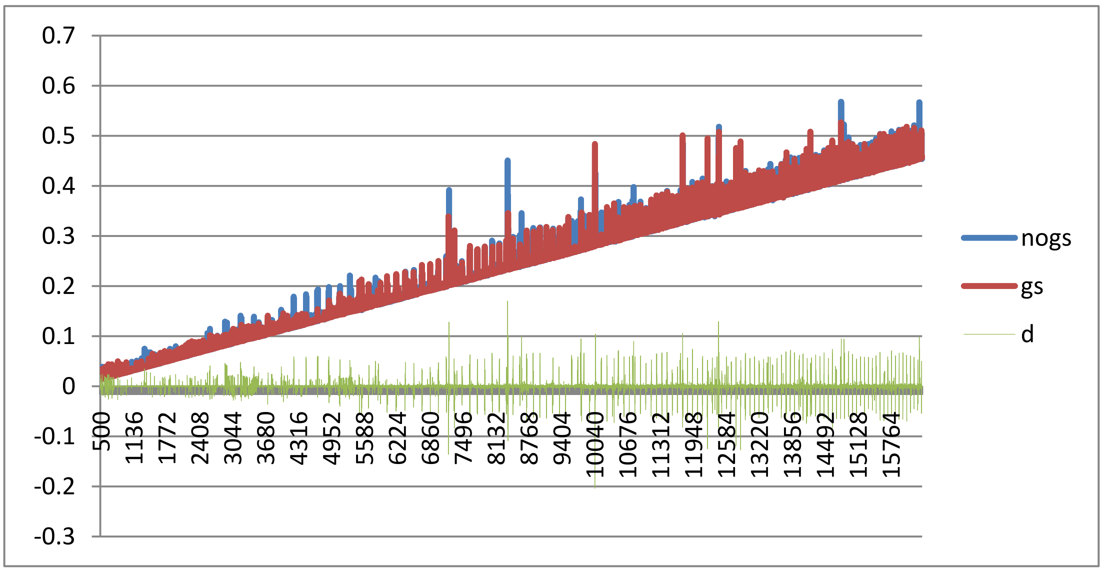
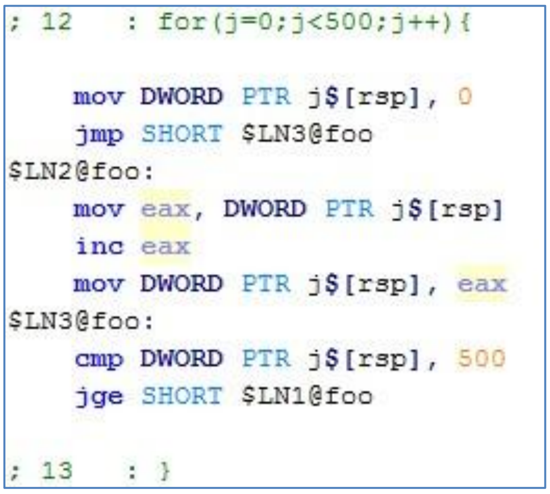


Figure 38 Data of Experiment 7

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. The green line is the absolute difference of execution time (d=nogs-gs).

The result seems normal. According the chart, ignoring those noises, the performance difference between programs with and without GS protection seems insensible. Looking at the assembly version below, as we expected, the for-loop with empty content is doing the loop without assignment.



```
; 12    : for(j=0;j<500;j++){

    mov DWORD PTR j$[rsp], 0
    jmp SHORT $LN3@foo
$LN2@foo:
    mov eax, DWORD PTR j$[rsp]
    inc eax
    mov DWORD PTR j$[rsp], eax
$LN3@foo:
    cmp DWORD PTR j$[rsp], 500
    jge SHORT $LN1@foo

; 13    : }
```

Figure 39 Empty For-loop

Hence, we are convinced that the for-loop itself does not trigger the phenomenon of better performance with GS protection than with GS protection.

#### 4.2.2.6 Analysis on the boundary of a memory page

Then, another question is what happens when the stack grows beyond the size of a memory page. According to the document[58] and [59] on Microsoft’s website, the Visual Studio compiler inserts a function named __chkstk() during the prologue of a function when the compiler has to allocate larger space than one page of memory can offer. This __chkstk() is not only a helper function, but also a stack-checking function. On the ability of checking the stack, the aim of __chkstk() is different from that of /GS mechanism. As described before, the /GS mechanism checks whether the buffer overflows corrupt the return addresses. But the __chkstk() checks the availability of the stack space and commit more memory. However, whether the __chkstk() can directly affect the performance of GS protection is not clear in my research. One guess is that the chkstk function may cause performance loss, but it can boost the performance when working with GS protection. This hypothesis will be verified in future work.

At this point of the research, we can see that the performance boost of GS protection is probably related to the memory paging mechanism. With the existence of security cookie, the stack of the function is enlarged, leading to faster access to the data on the buffer of the stack. Furthermore, the __chkstk() function is involved with the GS protection mechanism, which somehow affect the performance of programs.

#### 4.2.2.7 Experiments: With two arrays

Still, in the above functions, we have two variables: the size of the stack and the processed data. When the size of the buffer grows, not only does the stack grow, the times of assignment operations increase as well.

To split these variables, we can use two arrays, one of which is supposed to enlarge the size of the stack, while the other processing data. Fixing one of them, the effect of the other variable can be checked.

The function is like this:

```
void foo(){
        int dummy[N];
        int buffer[M];
        int j;
        for(j=0;j<M;j++){
                buffer[j]=1;}
}
```

The only use of the array “dummy” is to enlarge the stack size by increasing N. The array “buffer” is used to execute assignment operations on the stack.

When the value of M is set to a fixed value 100 and the N changes from 2 to 1509, the experiment result is as follows:

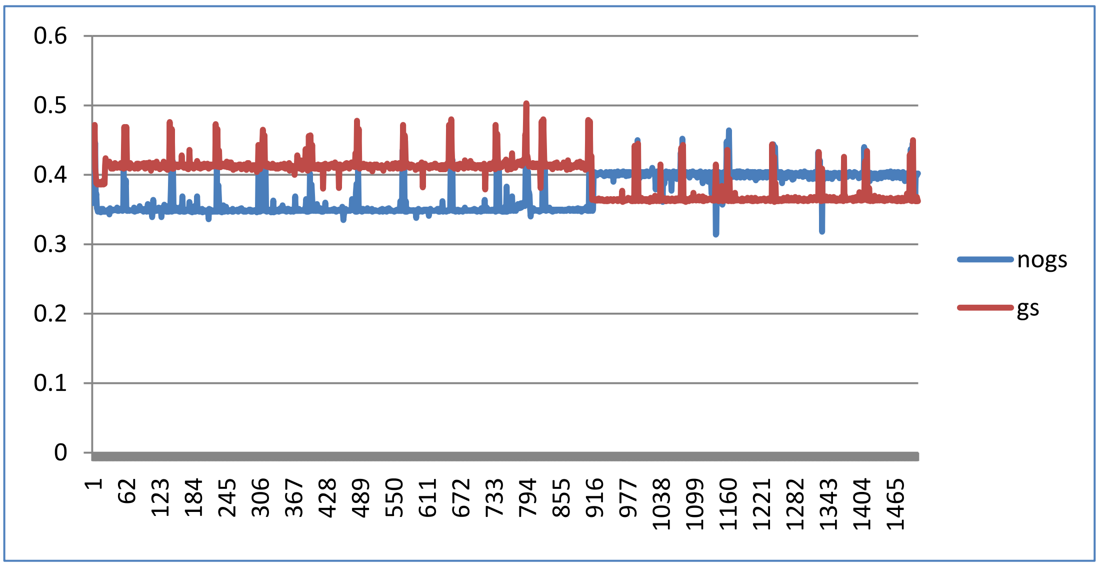


Figure 40 Data of Experiment 8

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. According to the chart, the no-GS version programs run faster than the GS protected version until the N reaches some turning point around 900. When N is greater than 900, the GS protected programs have better performance than the programs without GS protection. This turning point is where the stack of the function goes beyond the size of one memory page. It seems like that after the size of the stack exceeds the size of the page, the performance of assigning buffer array with GS protection is improved, but the no-GS version gets some performance loss.

When the value of M is set to a fixed value 200 and the N changes from 2 to 1509, the experiment result is as follows:

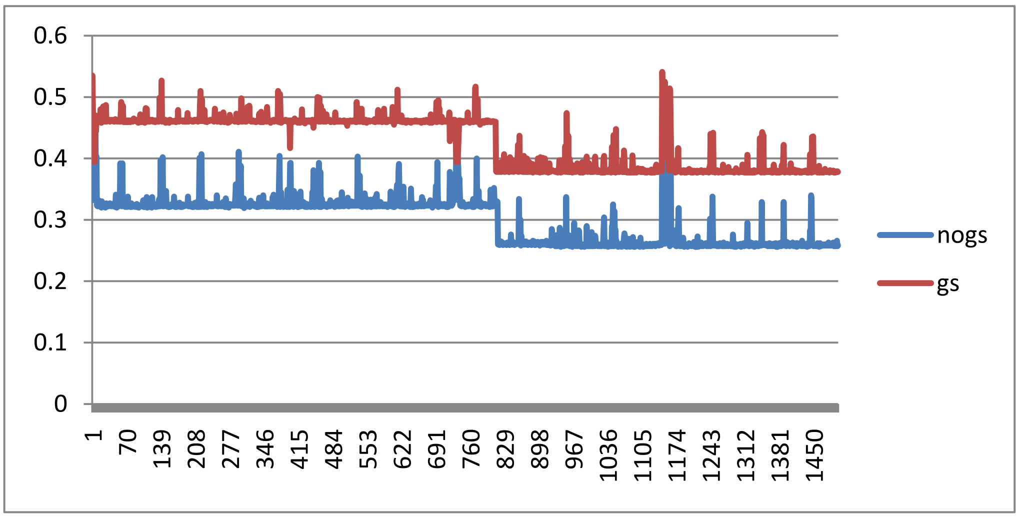


Figure 41 Data of Experiment 9

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. According to the chart, the no-GS version programs run faster than the GS protected version. There is still a turning point where the stack of the function goes beyond the size of one memory page. After that point, the performance of assigning buffer array with GS protection is remarkably improved, and the no-GS version also gets some performance boost.

The above two cases indicate that the performance of certain function can be improved when normally the performance will get degraded, after the size of the stack exceeds the page size. To some extent, this supports our hypothesis that the __chkstk() function working with GS protection leads to performance boost of certain functions.

Then, let the N be fixed as 350 and M change from 5 to 1000:

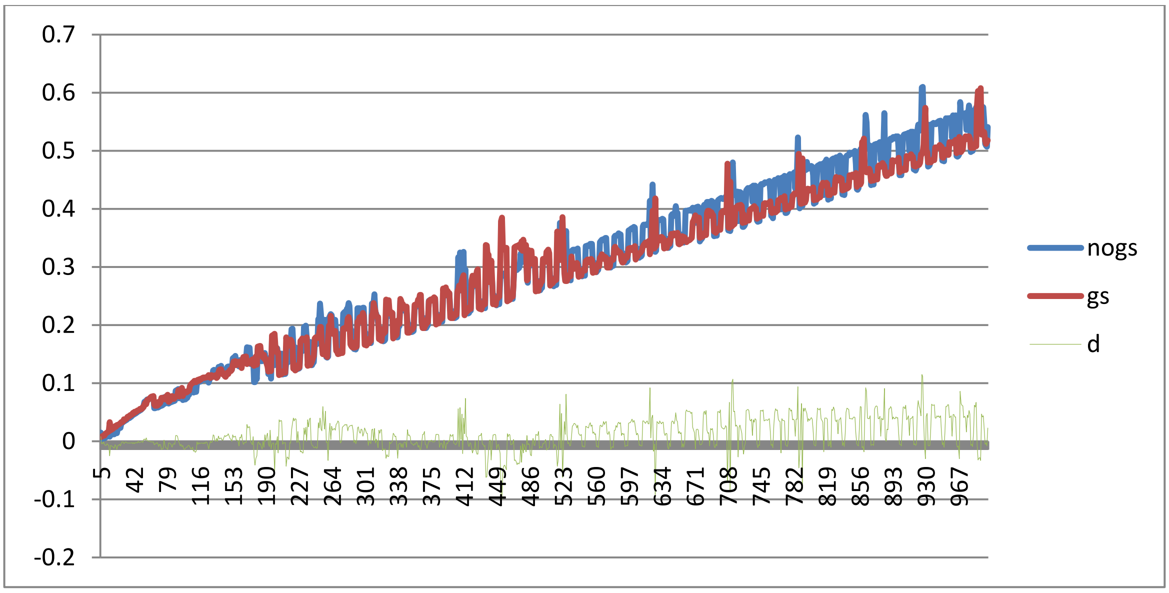


Figure 42 Data of Experiment 10

In this line chart, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. This time, the stack exceeds the page size when M is equal to about 550. Before that point, both versions have almost the same execution time. After that point, the no-GS version executes slower than the GS protected version, which is similar to the second kind of function we have talked.

### 4.2.3 Conclusion

In conclusion of the verifying of this hypothesis, we are convinced that the memory paging mechanism of Microsoft definitely has influence on the performance issue of programs with GS protection. When the size of a stack is within the page size, the normal operations such as assignment does not show much difference between with and without GS protection. But when the stack covers more than one page, the performance becomes abnormal. Furthermore, in our experiments, with the stack growing, the performance boost occurs regularly, matching the memory paging. All of those indicate that the memory paging mechanism may relate to the improvement with GS protection.

## 4.3 Hypothesis 2: Data alignment influences the performance

### 4.3.1 Basic idea

The second hypothesis is about the data alignment. Since the CPU accesses aligned data faster than unaligned, it is possible for the operations on that data with GS protection to show better performance than without GS, if the GS protection can somehow make the unaligned data aligned.

As we known, the computer has the ability to access a piece of data on the memory in chunks of certain size. On the 32-bit system, the chunk is 4 bytes, while it is 8 bytes on the 64-bit system. Hence, one the 32-bit system, the CPU can easily fetch the data whose offset is some multiple of 4. We say that the data aligned when its offset matches that case. If a piece of data is located at the address, whose offset is not any multiple of 4, it requires extra operation for the CPU to calculate the location.

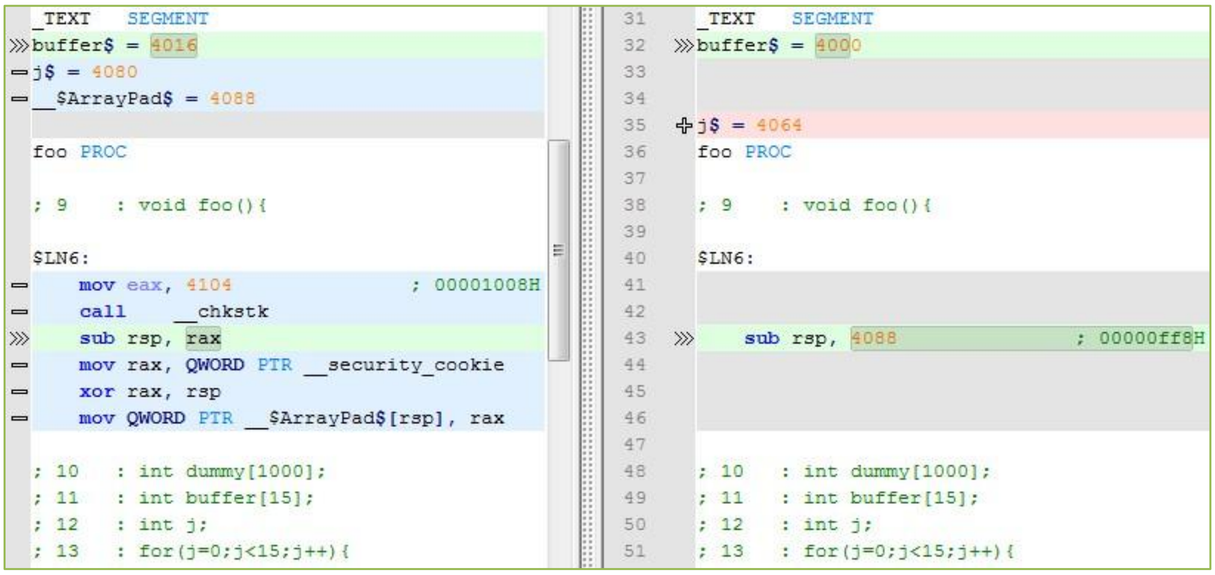
The typical well-known case of data alignment in compiler is the structure in C and C++. When we are using a structure type in C and C++, mixing different data types together, the size of this structure is not the sum of the sizes of those types, if the compiler is up-to-date. There is padding added to the types whose sizes are odd by the compiler.

On the function's calling stack, if the data is not aligned, the CPU also needs to perform extra calculation and fetch the data with more operations. Moreover, the size of buffer array in a function cannot always be a multiple of 4 or 8. Hence, chances are the array data is not aligned on the stack.

Therefore, we can guess that the adding extra data like the security cookie may change the alignment in the stack, leading to some changes to the unaligned data and then the performance gets boosted.

### 4.3.2 Validations

To study the effect of data alignment to the performance of programs, we can look at the assembly code of the some functions. As known to all, the stack is set up during the prologue of the function, so the prologue part of a function is selected and analyzed.



```
_TEXT   SEGMENT
buffer$ = 4016
j$ = 4080
__$ArrayPad$ = 4088

foo PROC

; 9     : void foo(){

$LN6:
        mov eax, 4104                           ; 00001008H
        call    __chkstk
        sub rsp, rax
        mov rax, QWORD PTR __security_cookie
        xor rax, rsp
        mov QWORD PTR __$ArrayPad$[rsp], rax

; 10    : int dummy[1000];
; 11    : int buffer[15];
; 12    : int j;
; 13    : for(j=0;j<15;j++){
```

```
_TEXT   SEGMENT
buffer$ = 4000



j$ = 4064
foo PROC

; 9     : void foo(){

$LN6:


        sub rsp, 4088                           ; 00000ff8H




; 10    : int dummy[1000];
; 11    : int buffer[15];
; 12    : int j;
; 13    : for(j=0;j<15;j++){
```

Figure 43 Comparison of two version of assembly code

The above picture is the comparison of the prologue parts of assembly codes with and without GS protection. These assembly codes are generated from the same C source code. On the left is the one with GS protection, while on the right is the one without GS protection. The values of "buffer$", "j$", "__$ArrayPad$" are the offsets of variables "buffer", "j" and "ArrayPad" to the stack pointer "rsp". The lines of "sub rsp, rax" and "sub rsp, 4088" are commands used to allocate stack space of certain size.

According to those offset values and stack sizes, we can figure out the layout of data on the stack. For example, on the left, the size of the stack is 4104 bytes and the buffer array is 4080-4016=64 bytes including padding. Then the dummy array is 4x1000=4000 bytes and does not need padding.

So, the layout of data in the stacks is as follows:

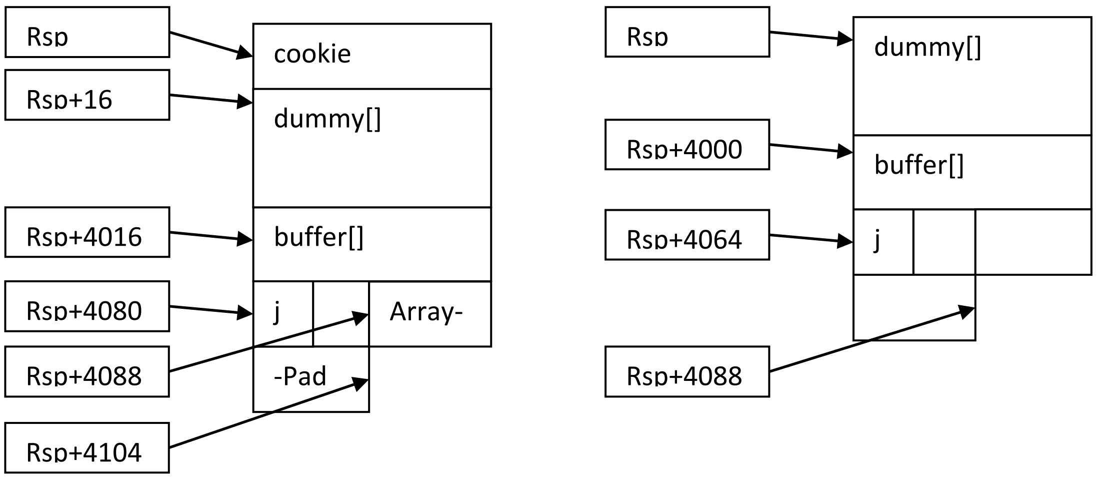


Figure 44 Alignment on the stack

According to the picture above, the local variables are always 16-byte aligned, because the source codes are compiled with the Visual Studio 2010 x64 compiler. They would be 8-byte aligned if using the 32-bit compiler tool. Furthermore, the GS protection only adds a 16-byte cookie, and that cookie is just aligned with other data.

Then, the size of buffer array is changed to see if it is always aligned. Results show that compiler adds padding space to the array to make its size dividable by 16. Therefore, the data is always aligned. Althrough the ArrayPad is not aligned, this piece of data is only accessed once for checking the security cookie and its effect on performance can be ignored.

### 4.3.3 Conclusion

In conclusion, this hypothesis is not confirmed. The data alignment with GS protection might trigger the performance boost, as the data are aligned and the security cookie does not affect the alignment of other data on the stack.

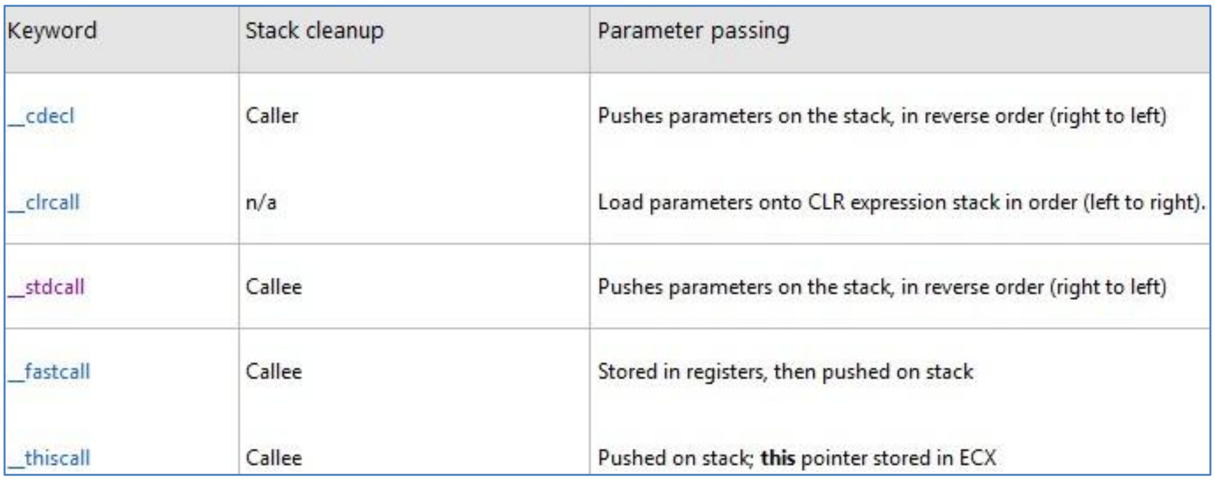
## 4.4 Hypothesis 3: Arguments passing influences the performance

### 4.4.1 Basic idea

This hypothesis is about the argument passing. Considering the functions with arguments, the arguments are placed very close to the return address and security cookies on the stack. Thus, there was some probability that the GS protection can affect the performance of arguments passing and accessing, which can somehow boost the overall performance.

In general, when a function is called with some arguments, those arguments are first put into the registers and then passed to the stack. For example, on the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, only four registers are used, which means that arguments more than four have to be stored in the stack [60]. Therefore, the situations in dealing with those arguments can be complicated with the existence of GS protection's cookies.

The calling conventions on Windows platform set a standard way of argument passing. These calling conventions are different between 32-bit and 64-bit systems. They specify how the arguments are passed (order, size), where they are stored (register or stack), and who does the cleanup (caller or callee).[61] Actually, those conventions define the prologue and epilogue of the function, so it is natural to guess that with different amount of arguments passing to the function, the performance may differ between with GS and without GS protection, due to the different way of storing those arguments.



| Keyword | Stack cleanup | Parameter passing |
|---|---|---|
| __cdecl | Caller | Pushes parameters on the stack, in reverse order (right to left) |
| __clrcall | n/a | Load parameters onto CLR expression stack in order (left to right). |
| __stdcall | Callee | Pushes parameters on the stack, in reverse order (right to left) |
| __fastcall | Callee | Stored in registers, then pushed on stack |
| __thiscall | Callee | Pushed on stack; **this** pointer stored in ECX |

Figure 45 Calling conventions on x86 Taken from [62]

### 4.4.2 Validations

To verify the third hypothesis, we need to figure out whether the GS protection working with arguments passing can show better performance than the same program without GS. Thus, some arguments should be set and passed to the functions. Through changing the number of the arguments, the programs may act in different ways: passing arguments by registers or stack. With the existence of security cookie, the argument passing mechanism may improve the performance of programs. This is a hypothesis, and we need to construct functions taking different number of arguments.

Therefore, the following function is set:

```
void foo(int i0,...,int in){

int buffer[n+1];

buffer[0]=i0;

…

buffer[n]=in;

}
```

Python is used again to generate a lot of source files which contain different constant integer as n. Then, those source files are compiled and run. The result is as below:

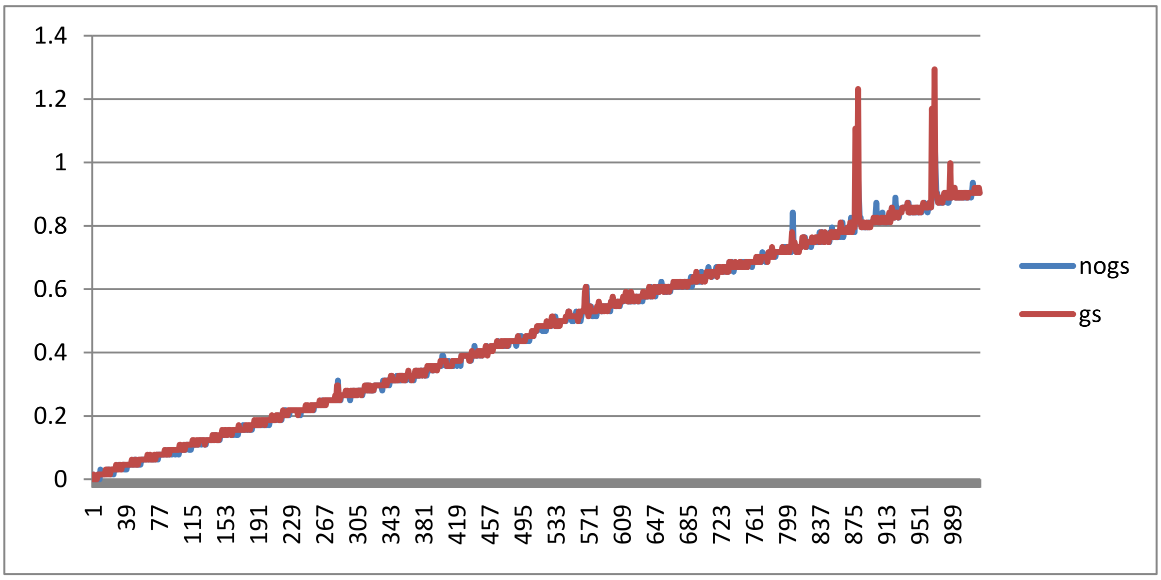


Figure 46 Data of Experiment 11

In this graph, the vertical axis is the execution time and the horizontal axis shows the size of the buffer (N). The blue line represents the programs without GS protection, while the red line shows the programs with GS protection. This line chart does not show much difference between two versions of programs, meaning the argument passing may not boost the performance with GS protection. So, this hypothesis is not proved correct.

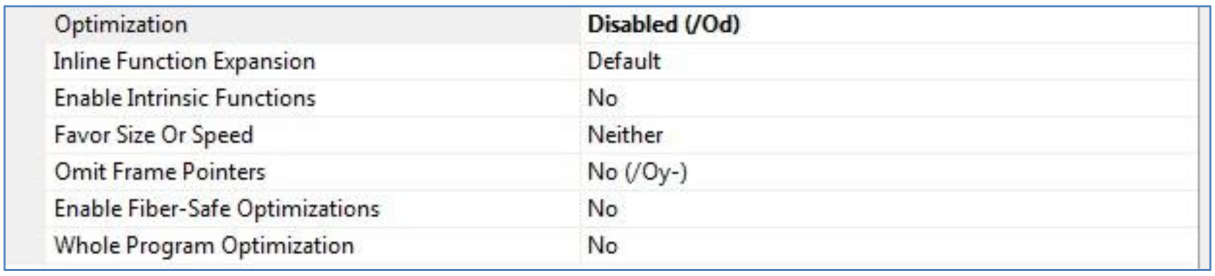
### 4.4.3 Conclusion

In conclusion, according to the above experimental results and analysis, this hypothesis is not proved.

## 4.5 Hypothesis 4: Inlining influence the performance

### 4.5.1 Basic idea

There are a number of optimization options in Visual Studio, such as inline function expansion, intrinsic functions, favour size or speed, omit frame pointers, fiber-safe optimizations and whole program optimization, etc[51].



| | |
|---|---|
| Optimization | **Disabled (/Od)** |
| Inline Function Expansion | Default |
| Enable Intrinsic Functions | No |
| Favor Size Or Speed | Neither |
| Omit Frame Pointers | No (/Oy-) |
| Enable Fiber-Safe Optimizations | No |
| Whole Program Optimization | No |

Figure 47 A snapshot of Optimizations in Visual Studio 2010

The options of the compiler may interfere themselves, so there is certain possibility for some of the above optimization options to interfere the application of GS protection, somehow resulting in better performance than without GS protection. This could happen if a piece of code shows worse performance with certain optimization than without it.

Inlining is just the kind of optimization we were looking for.

Inlining is an optimization to remove the prologue and epilogue of a function. To do this, the compiler inserts the function subroutine into the main routine, instead of calling the function in the main routine. Doing so, the overhead of prologue and epilogue is reduced and this function is not pushed into the calling stack for this time.

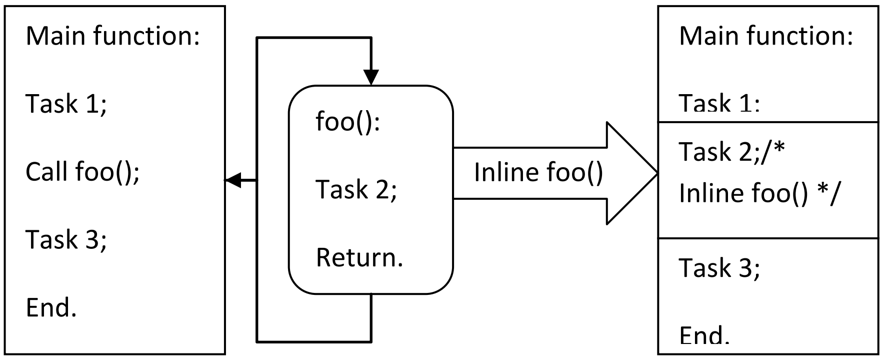


Figure 48 Inlining

As the inline function does not create stack frame for itself when called, there is no need for stack protection. Therefore, the inlining optimization does not coexist with the GS protection. The two mechanisms are mutually exclusive.

According to [63], there are some shortcomings of this type of optimization. Those problems such as register pressure and cache slowdown are able to cause performance loss. So, there is a guess that programs with inlining may cause larger performance loss than programs with GS protection. When this hypothesis becomes true in certain situation, the programs may seem to have better performance with GS protection than without GS protection. The verification of this hypothesis will be talked about in next chapter.

### 4.5.2 Validation

To verify whether inlining can cause more performance loss than the GS protection, we need to construct some functions that have better performance without inlining than with inlining.

The “inline” keyword is added to the function foo(), and the pseudo code of the function looks like:

```
inline int foo(){
    Local variables;
    Processing tasks;
    Return;
}
```

However, after compiled, the source code generated assembly code in which the foo function is not inlined.

Then, more research has been done to the inlining in the Visual Studio. It turns out that the “inline” keyword does not force the function inlined. It is only a suggestion to the compiler. The compiler decides whether the function is to be inlined or not [64]. According to [64], there are many restrictions on what kind of functions can be inlined. Normally, functions that contain lots of variables and huge amount of data are not inlined. Meanwhile, as talked previously, with few operations and small size buffer, the testing will not show obvious difference between programs with GS and without GS protection. Thus, the chance is quite small to find a function which can be inlined and GS protected, and those two versions (inlined and GS protected) can show remarkable difference on performance.

### 4.5.3 Conclusion

In conclusion of this part, the inlining might not cause the phenomenon of better performance with GS protection than without it.  According to our analysis, the Visual Studio compiler can hardly be able to inline a piece of code containing GS buffer. Even if there is a piece of code that meets the requirement, the difference between its two versions may be small, because inlined functions are usually small.

## 4.6 Hypothesis 5: Caching influences the performance

### 4.6.4 Basic idea

The last but not the least hypothesis is about caching. The cache is a kind of the high speed memory, which normally lies between CPU and the main memory. It is expensive but faster than the main memory, so it is used to store the most frequently used data. Since the data is divided into blocks, the cache loads data as one block. The cache only loads data when the CPU cannot find data in the cache, which is named a cache miss. If the data is found on the cache, it is called a cache hit. [65] Therefore, the cache can influent the performance of programs with its hit ratio.

The stack, which is on the main memory, can be benefited by the cache, too. Hence, it is interesting to find out whether caching mechanism can contribute the better performance without GS protection than without GS protection.

### 4.6.5 Validation

Verification of this hypothesis is not completed. Since the cache is difficult to monitor and control, more time and knowledge is required for researching the relation between the caching and the phenomenon we are studying.

According to [65], parameters of caches such as capacity, block size, associativity, and replacement policy, should be known. Based on those parameters, experiments could be designed to check the behaviour of the cache when programs with and without GS protection are running.

### 4.6.6 Conclusion

So, the future work may try to find some clue from the caching behaviours of programs with and with GS.

## 4.7 Overall Analysis

In the chapter 3, we are convinced that some programs with GS protection may have better performance than without GS protection. Then, in the previous chapter, we have listed five potential aspects, with which figure out the reasons for that phenomenon. In this chapter, research and experiments have been done from the five aspects. The research shows that the memory paging, cooperating with GS protection, may boost the performance.

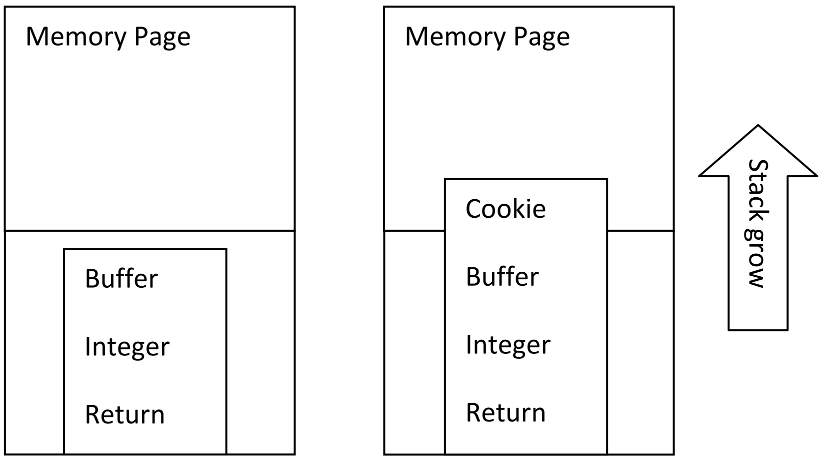


Figure 49 Stack and Memory pages

Although four other aspects show no sign of contributing to the performance boost with GS protection, we still cannot rule out the possibility that they working with other factors may contribute to that phenomenon. For example, the way of caching within the memory page may affect the performance with GS protection.

Since performance issue is always involving various factors, it is hard to say any of the aspects solely causes the phenomenon. Moreover, sometimes one factor might improve the performance a little bit but not enough to offset the performance loss of the GS protection. Then, several factors working together may improve the performance to the extent that the programs with GS protection seems to run faster than without it.

# Chapter 5. Conclusion and Future work

## 5.1 Conclusion

In this project, the knowledge relating to fields such as the buffer overflow and stack protection techniques was studied. Then, we explored the stack protection mechanism of Microsoft’s platform, GS protection. With the inspiration of my supervisors, experiments were done to check the performance of the OpenSSL with and without GS protection. The results of the OpenSSL speed experiments show that the programs in some cases may have a better performance with GS protection than without it. This phenomenon is counter-intuitive.

To figure out the potential reasons for this phenomenon, we carried out more research and experiments from five aspects. In those experiments, pieces of batch script and Python code were written to generate large number of C code files, compile them, and execute them, and process the outcome data. After that, it could be seen that the memory paging, cooperating with GS protection, may boost performance, except for the situation of performance within one memory page, which are still in need of being figured out.

In conclusion, during the whole project, following goals were achieved:

- We have surveyed the techniques in buffer overflow attacks and their countermeasures.
- We have looked into Microsoft’s stack protection technique GS.
- We have evaluated the usage of stack protection techniques and the performance of some software programs from experiments.
- We have listed five hypothesises to explain how techniques like GS really affect the performance of the outcome programs. One hypothesis, memory paging, has been emphasized and validated.

## 5.2 Future work

Although four other aspects show no sign of contributing to the performance boost with GS protection, they still cannot rule out the possibility that they work with other factors that may contribute to the phenomenon. For example, the way of caching within the memory page may affect the performance with GS protection. Since performance issues always involve various factors, it is hard to say whether any of the aspects solely causes the phenomenon. Moreover, sometimes one factor might improve performance a little but not enough to offset the performance loss of the GS protection. Thus, several factors working together may improve the performance to the extent that the programs with GS protection seem to run faster than without it. Therefore, future work could be done focusing on the combination of several factors.

Work figuring out how the GS protection improves performance in certain situations, is significant to the compiler techniques. Future work may also be done on adjusting the compiler to generate more efficient machine code according to factors like the size of stack with GS cookies.

Furthermore, according to the experiment in chapter 3, similar stack protection techniques such as GCC's ProPolice may cause the same performance boost as the GS does. Future work can also be carried out with such stack protections.

## References:

1. Cowan, C., et al. *Buffer overflows: attacks and defenses for the vulnerability of the decade*. in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*. 2000.
2. CERT. *CERT Advisory CA-2003-20 W32/Blaster worm*. 2003; Available from: http://www.cert.org/advisories/CA-2003-20.html.
3. Microsoft. *Security Bulletin*. Available from: http://www.microsoft.com/technet/security/current.aspx.
4. Tuck, N., B. Calder, and G. Varghese, *Hardware and binary modification support for code pointer protection from buffer overflow.* Micro-37 2004: 37th Annual International Symposium on Microarchitecture, Proceedings, 2004: p. 209-220.
5. Hennessy, J.L. and D.A. Patterson, *Computer architecture : a quantitative approach*. 3rd ed. 2003, Amsterdam ; London: Morgan Kaufmann. xxxi, 883, 87, 42, 22, 44 p.
6. Knuth, D.E., *The art of computer programming*. 2005, Upper Saddle River, NJ ; London: Addison-Wesley. v.
7. Park, S.H., et al., *The dynamic buffer overflow detection and prevention tool for Windows executables using binary rewriting.* 9th International Conference on Advanced Communication Technology: Toward Network Innovation Beyond Evolution, Vols 1-3, 2007: p. 1776-1781.
8. Microsoft, *STACKFRAME64 Structure.*
9. *Machine Language for Beginners*. Available from: http://www.atariarchives.org/mlb/introduction.php.
10. Wikipedia. *Stack buffer overflow*. Available from: http://en.wikipedia.org/wiki/Stack_buffer_overflow.
11. SecurityFocus. *CORE-2007-0219: OpenBSD's IPv6 mbufs remote kernel buffer overflow*. 2007; Available from: http://www.securityfocus.com/archive/1/462728/30/150/threaded.
12. Wikipedia. *Buffer overflow*. Available from: http://en.wikipedia.org/wiki/Buffer_overflow.
13. Koziol, J., et al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. 2004: John Wiley \& Sons.
14. One, A. *Smashing The Stack For Fun And Profit*. Available from: http://www.phrack.com/issues.html?issue=49&id=14#article.
15. Monica Chew, D.S., *Mitigating Buffer Overflows by Operating System Randomization*. 2002, CMU-CS.
16. Foster, J.C., M. Price, and ebrary Inc., *Sockets, shellcode, porting & coding reverse engineering exploits and tool coding for security professionals*. 2005, Syngress Publishing: Rockland, Mass.
17. Arce, I., *The shellcode generation.* IEEE Security & Privacy, 2004. **2**(5): p. 72-76.
18. Mason, J., et al., *English shellcode*, in *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, ACM: Chicago, Illinois, USA. p. 524-533.
19. Zhang, K., T. Zhang, and S. Pande, *Memory protection through dynamic access control.* MICRO-39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006: p. 123-134.
20. Eclipse, S. *Honeynet Project Scan of the Mohth for April 2002*. 2002; Available from: http://www.phreedom.org/solar/honeynet/scan20/scan20.html.
21. Jonathan Pincus, B.B., *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns.* IEEE Security & Privacy, 2004. **2**(4): p. 20-27.
22. Sabelfeld, A. and A.C. Myers, *Language-based information-flow security.* Ieee Journal on Selected Areas in Communications, 2003. **21**(1): p. 5-19.

23. Chiueh, T.C. and F.H. Hsu, *RAD: A compile-time solution to buffer overflow attacks.* 21st International Conference on Distributed Computing Systems, Proceedings, 2001: p. 409-417.
24. Ashcraft, K. and D. Engler, *Using programmer-written compiler extensions to catch security holes.* 2002 Ieee Symposium on Security and Privacy, Proceedings, 2002: p. 143-159.
25. Thomas Plum, D.M.K., *Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool.* ACM, 2005.
26. Michael Zhivich, T.L., Richard Lippmann, *Dynamic Buffer Overflow Detection.* BUGS: Workshop on the Evaluation of Software Defect Detection Tools, 2005.
27. Chess, B. and G. McGraw, *Static analysis for security.* IEEE Security & Privacy, 2004. **2**(6): p. 76-79.
28. Puchkov, F.M. and K.A. Shapchenko, *Static analysis method for detecting buffer overflow vulnerabilities.* Programming and Computer Software, 2005. **31**(4): p. 179-189.
29. Leroy, X., *Computer security from a programming language and static analysis perspective.* Programming Languages and Systems, 2003. **2618**: p. 1-9.
30. Vinod Ganapathy, S.J., David Chandler, David Melski, David Vitek. *Buffer overrun detection using linear programming and static analysis*. in *Conference on Computer and Communications Security*. 2003: ACM.
31. Dor, N., M. Rodeh, and M. Sagiv, *CSSV: Towards a realistic tool for statically detecting all buffer overflows in C.* Acm Sigplan Notices, 2003. **38**(5): p. 155-167.
32. Larochelle, D. and D. Evans, *Statically detecting likely buffer overflow vulnerabilities.* Usenix Association Proceedings of the 10th Usenix Security Symposium, 2001: p. 177-189.
33. Xie, Y., A. Chou, and D. Engler, *ARCHER: using symbolic, path-sensitive analysis to detect memory access errors*, in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 2003, ACM: Helsinki, Finland. p. 327-336.
34. McMaster, S. and A. Memon, *Call Stack Coverage for GUI Test-Suite Reduction*, in *Proceedings of the 17th International Symposium on Software Reliability Engineering*. 2006, IEEE Computer Society. p. 33-44.
35. Del Grosso, C., et al., *Detecting buffer overflow via automatic test input data generation.* Computers & Operations Research, 2008. **35**(10): p. 3125-3143.
36. OpenBSD, *OpenBSD Security.*
37. Cowan, C., et al., *StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks*, in *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*. 1998, USENIX Association: San Antonio, Texas. p. 5-5.
38. Microsoft. *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. 2006; Available from: http://support.microsoft.com/kb/875352.
39. Ozdoganoglu, H., et al., *SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address.* IEEE Trans. Comput., 2006. **55**(10): p. 1271-1285.
40. Xu, W., D.C. DuVarney, and R. Sekar, *An efficient and backwards-compatible transformation to ensure memory safety of C programs.* SIGSOFT Softw. Eng. Notes, 2004. **29**(6): p. 117-126.
41. Peter Silberman, R.J., *A Comparison of Buffer Overflow Prevention Implementations and Weaknesses.* 2004.
42. Litchfield, D. *Buffer underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows Platform*. 2005; Available from: http://www.ngssoftware.com/papers/xpms.pdf.
43. Tsai, T. *A Discussion of Performance Optimizations for Compiler-Based Buffer Overflow Instrumention*. in *Supplemental Proceedings of the International Conference on Dependable Systems and Networks*. 2006.

44. Dhurjati, D. and V. Adve, *Backwards-compatible array bounds checking for C with very low overhead*, in *Proceedings of the 28th international conference on Software engineering*. 2006, ACM: Shanghai, China. p. 162-171.
45. Younan, Y., et al., *Extended protection against stack smashing attacks without performance loss.* 22nd Annual Computer Security Applications Conference, Proceedings, 2006: p. 429-438.
46. Qi, H. *GS*. 2009; Available from: http://blogs.msdn.com/vcblog/archive/2009/03/19/gs.aspx.
47. MSRC. *Enhanced GS in Visual Studio 2010*. 2010; Available from: http://blogs.technet.com/srd/archive/2009/03/20/enhanced-gs-in-visual-studio-2010.aspx.
48. Whitehouse, O., *Analysis of GS protections in Microsoft Windows Vista*. 2007, Symantec Advanced Threat Research.
49. Microsoft. *BinScope Binary Analyzer*. 2009; Available from: http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=90e6181c-5905-4799-826a-772eafd4440a.
50. Microsoft, */GS (Buffer Security Check).* 2010.
51. Microsoft. */O Options (Optimize Code)*. 2010; Available from: http://msdn.microsoft.com/en-us/library/k1ack8f1.aspx.
52. *Openssl: The Open Source toolkit for SSL/TLS*. Available from: http://www.openssl.org/.
53. *OpenSSL: Documents, speed*. Available from: http://www.openssl.org/docs/apps/speed.html.
54. *GCC extension for protecting applications from stack-smashing attacks*. Available from: http://www.trl.ibm.com/projects/security/ssp/.
55. Microsoft. *Managing Virtual Memory*. 1993; Available from: http://msdn.microsoft.com/en-us/library/ms810627.aspx.
56. Microsoft. *Virtual Address Space*. Available from: http://msdn.microsoft.com/en-us/library/aa366912%28VS.85%29.aspx.
57. Microsoft. *Paging*. Available from: http://en.wikipedia.org/wiki/Paging.
58. Microsoft, *Description of the stack checking for Windows NT-based applications.* 2005.
59. Microsoft, *Prolog and Epilog.* 2010.
60. *Understanding the Stack*. 2003; Available from: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html.
61. Microsoft. *Calling Conventions*. Available from: http://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx.
62. Microsoft. *Argument Passing and Naming Conventions*. 2010; Available from: http://msdn.microsoft.com/en-us/library/984x0h58.aspx.
63. Wikipedia, *Inline expansion.*
64. Microsoft. *inline,_inline,_forceinline*. 2010; Available from: http://msdn.microsoft.com/en-us/library/z8y1yy88.aspx.
65. LaMarca, A. and R.E. Ladner, *The influence of caches on the performance of sorting*, in *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. 1997, Society for Industrial and Applied Mathematics: New Orleans, Louisiana, United States. p. 370-379.