國立交通大學

資訊科學與工程研究所

碩 士 論 文

快速偵測與恢復軟體溢位錯誤之方法

An Efficient Method for Buffer Overflow Detection and Recovery

NCTU watermark

研 究 生：賴怡良

指導教授：吳毅成 教授

中 華 民 國 九 十 五 年 七 月

快速偵測與恢復軟體溢位錯誤之方法

An Efficient Method
for Buffer Overflow Detection and Recovery

研 究 生：賴怡良　　Student：Yi-Liang Lai

指導教授：吳毅成　　Advisor：I-Chen Wu

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

July 2006

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 五 年 七 月

快速偵測與恢復軟體溢位錯誤之方法

學生：賴怡良　　　　　　　　　　　　　　指導教授：吳毅成博士

國立交通大學資訊科學與工程研究所碩士班

網際網路技術與應用實驗室

摘要

緩衝區溢位(buffer overflow)是一個常見的軟體漏洞，現今許多應用程式都發現有過這種漏洞，因而造成嚴重的安全問題。為了要能夠有效地解決緩衝區溢位問題，研究者提出各種緩衝區溢位的偵測與恢復機制，而這些機制所造成的額外效能負擔多寡是這些機制是否實用的關鍵。此外，我們經由實驗得知，一個沒有恢復能力的偵測機制，受其保護的程式在遭遇攻擊時，服務的能力將大幅度下降，所以我們的研究中專注在有效率的緩衝區溢位偵測與恢復機制上。

目前被認為偵測範圍最廣的 CRED (C Range Error Detector) 偵測技術與以 CRED 為基礎的 BMB (Boundless Memory Blocks) 恢復技術，某些情形下會造成執行效能相較原程式約 30 倍的低落。本論文提出一個名叫 BODAR(Buffer Overflow Detection And Recovery)的方法，能夠在執行時期利用作業系統的分頁保護機制，以事件驅動的方式正確且有效率地偵測緩衝區溢位的發生。此外，BODAR 離散地配置各個緩衝區，使得各緩衝區後都有一塊未配置區域能夠用來容忍溢位的資料，進行恢復時只需直接增長緩衝區至未配置區域。如此除了能得到與 CRED 近乎相同的偵測與恢復能力以外，亦能有相當優良的效能表現。我們的實驗顯示有 BODAR 保護的程式只比原來的程式多增加了 10%到 80%的執行時間。

An Efficient Method for Buffer Overflow Detection and Recovery

Student: Yi-Liang Lai Advisor: I-Chen Wu

Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

Abstract

Buffer overflow is a kind of common software vulnerabilities. Today, it exists in many applications and therefore causes serious security problems. In order to solve such problems, many researchers have proposed mechanisms for buffer overflow detection and recovery. On one hand, the practicability of these mechanisms could be primarily decided by the amount of overhead in runtime. On the other hand, according to our experiments, when attacking a server with the detection ability but without recovery ability, we found that its service availability would be degraded significantly. Based on the two observations, our research is focused on an efficient buffer overflow detection and recovery mechanism.

Two past solutions, CRED(C Range Error Detector) and BMB(Boundless Memory Blocks), have been recognized as the best protection method in overflow-detecting and the most practical recovery method base on CRED, respectively. However, in certain situations, both could result in lower performance by a factor of 30. In this paper, we proposed a method named BODAR for the buffer overflow problem. BODAR uses the page protection mechanism of OS which is event-driven to detect buffer overrun correctly and efficiently. BODAR also sparsely setup an unallocated region behind each buffer. The unallocated regions would be allocated for tolerance of out-of-bound data when recovering. Our experiments showed that the execution time of BODAR-enable programs was only 10% to 80% slower than that of non-protected ones.

誌謝

首先我要感謝我的指導教授，吳毅成老師，在我二年的研究生生活中不斷指導我做學問的態度與方法，讓我能夠不斷的改進自己的缺點，積極向前。在我寫這篇論文時，更給我許多寶貴的意見及指導，使這篇論文能夠順利的完成。

另外還要感謝黃世昆老師與徐健智學長，在研究過程中，他們不斷修正我粗陋的想法，並提供了我許多新穎的觀點與精闢的見解，讓我對許多問題能夠迎刃而解。

謝謝汪益賢學長，教導我如何簡單明確的表達想法，並指正我研究中的缺失與不足的地方。謝謝實驗室的同學育嘉、承翰和俊彬在這兩年中同甘共苦、相互勉勵。謝謝我大學以來的室友事修，他總是我第一個口頭報告的對象。謝謝學弟君鴻、炯斑老是幫忙不過來的我買便當。

最後，謝謝老爸總是叮嚀我不要太勞累，謝謝老媽總是在我回家時熬一堆補品，謝謝妹妹在我不在家時能幫爸媽分擔辛勞。因為有這麼可愛的家庭，使我能專心致力於我的研究，希望我的努力能給予他們小小的回報。

謹將此論文獻給所有幫助過我的人與我最愛的家人。

賴怡良 謹致

2006/7/16 于交通大學

目錄

摘要 i

Abstract ii

誌謝 iii

圖表目錄 vi

第一章　緒論 1

1.1 緩衝區溢位攻擊 1

1.2 相關緩衝區溢位防禦工具探討 3

1.2.1 原始程式碼分析技術 3

1.2.2 執行期溢位偵測技術 3

1.2.3 執行期溢位恢復技術 5

1.2.4 資料位址混亂技術(Obfuscation) 6

1.3 研究目標與方向 6

1.4 論文大綱 7

第二章　設計與實作 8

2.1 緩衝區的組織與配置 9

2.2 錯誤位址解析 12

2.3 溢位恢復處理 13

2.4 緩衝區的轉換 14

2.4.1 堆積緩衝區的轉換 14

2.4.2 堆疊緩衝區的轉換 15

2.4.3 靜態緩衝區的轉換 16

第三章　實驗與分析 18

3.1 偵測能力 18

3.2 恢復能力 19

3.2.1 Apache 19

3.2.2 thttpd 21

3.3 效能評估 22

3.3.1 重視輸出輸入(I/O-bound)的應用程式 23

3.3.2 重視 CPU 運算(CPU-bound)的應用程式 24

第四章　討論與改進 28
4.1 可使用的虛擬位址空間 28
4.2 作業系統的記憶體分頁管理機制對 BODAR 的影響 29
4.2.1 Page Reclaim 對效能的影響 30
4.2.2 TLB(Translation Look-aside Buffer)對效能的影響 31
4.3 減低記憶體空間浪費的折衷方法 32
第五章　結論與未來發展 35
參考文獻 37

圖表目錄

表 1-1　2000 年至 2005 年緩衝區溢位漏洞調查 1
表 1-2　一些易被誤用的 C 標準字串處理函式 2
圖 1-1　具有漏洞的程式碼及受到攻擊時記憶體的變化 2
表 1-3　一些執行期溢位保護機制的比較 4
圖 2-1　一個 FreeBSD 程序對虛擬記憶體空間的使用狀況 9
圖 2-2　BODAR 緩衝區的組織方式 10
圖 2-3　BODAR 緩衝區的配置方式 11
圖 2-4　堆疊緩衝區修改前後的程式碼 15
圖 2-5　靜態緩衝區修改前後的程式碼 17
表 3-1　一些具有緩衝區溢位漏洞的應用程式 19
圖 3-1　Apache 與 Apache-BODAR 受到攻擊時服務效能的比較。(a)連線成功率，(b)輸出量，(c)實際存取網頁數目，(d)平均回應時間。 20
圖 3-2　thttpd 與 thttpd-BODAR 受到攻擊時服務效能的比較。(a)連線成功率，(b)輸出量，(c)實際存取網頁數目，(d)平均回應時間。 22
表 3-2　Apache 與 thttpd 在無攻擊時，服務效能的比較 23
表 3-3　Apache 與 thttpd 的守衛區域大小比較 24
圖 3-3　gnupg 的實驗結果。(a)不同版本的 gnupg 用 RSA 加密六個檔案的結果，(b)不同版本的 gnupg 用 RSA 解密六個檔案的結果。 25
圖 3-4　gawk 的實驗結果。(a) gawk 執行 A 腳本程式，(b) gawk 執行 P 腳本程式，(c) gawk 執行 S 腳本程式。 26
圖 4-1　BODAR 在 IA-32 與 AMD-64 中虛擬記憶體空間使用情況 28
圖 4-2　不同版本 gawk 在執行過程中的 page reclaim 次數統計 30
圖 4-3　虛擬位址對應實體位址的轉換機制 31
圖 4-4　緩衝區溢位漏洞的特性分析。(a) 自 2002 年至 2005 年 SecuriTeam 的漏洞特性分析圖。(b) 2005 年 CVE 的漏洞特性分析圖。 33
圖 4-5　gawk 以 BODAR 折衷方式保護與其他版本的結果比較。 33
圖 4-6　gawk 以 BODAR 整合方式保護與其他版本的結果比較。 34
圖 5-1　未來可能的改進方法 36

第一章 緒論

C/C++是種相當被廣泛使用在各方面的程式語言，許多重要的程式甚至是作業系統都以其寫成，最主要的原因是著眼於C/C++優良的效率與強大的控制資源的能力。這種能力一旦沒有良好的處理，很容易在程式中引入潛在的漏洞，因而造成嚴重的後果。在這些漏洞中，稱作「緩衝區溢位(buffer overflow)」的漏洞是最為常見與嚴重的。根據我們調查CVE(Common Vulnerabilities and Exposures) [31]與SecuriTeam[4]的結果，一般來說緩衝區溢位漏洞與全部漏洞的比率在過去幾年都佔了20%左右，見表1-1。

表 1-1　2000 年至 2005 年緩衝區溢位漏洞調查

Year	CVE	SecuriTeam
2000	25.54%	6.81%
2001	23.56%	12.78%
2002	21.33%	12.70%
2003	29.31%	18.85%
2004	19.80%	20.23%
2005	14.22%	19.45%

本論文研究緩衝區溢位發生的原因，並比較過去其他研究所提出的方法，試著提出一套機制能夠快速地、有效率地解決緩衝區溢位問題。

1.1 緩衝區溢位攻擊

由於C/C++並沒有定義記憶體資料間的存取限制，當程式試圖去存取超出緩衝區邊界的資料時，並不會發出任何警告，儘管部分的處理器提供邊界檢查的指令，例如IA-32中的bound指令[13]，但大多數的C/C++編譯器不會特別為這些不知是否會發生溢位的記憶體存取產生程式碼去檢查，導致溢位存取得以在任何人都無法查知的情況下悄悄進行。

C/C++不處理就必須由程式設計師來處理。但C/C++不嚴謹的語言定義，程式設計師不易察覺是否在程式中隱含了溢位存取，再加上以錯誤的

態度使用設計不當的 C 標準字串函式庫（如表 1-2），故在現今的程式碼中存在有不少的緩衝區溢位漏洞。

表 1-2　一些易被誤用的 C 標準字串處理函式

Function prototype	Potential problem
`strcpy(char *dest, const char *src)`	May overflow the `dest` buffer.
`strcat(char *dest, const char *src)`	May overflow the `dest` buffer.
`getwd(char *buf)`	May overflow the `buf` buffer.
`gets(char *s)`	May overflow the `s` buffer.
`fscanf(FILE *stream, const char *format, ...)`	May overflow its arguments.
`scanf(const char *format, ...)`	May overflow its arguments.
`realpath(char *path, char resolved_path[])`	May overflow the `path` buffer.
`sprintf(char *str, const char *format, ...)`	May overflow the `str` buffer.

資料來源：Transparent Run-Time Defense Against Stack Smashing Attacks[3]

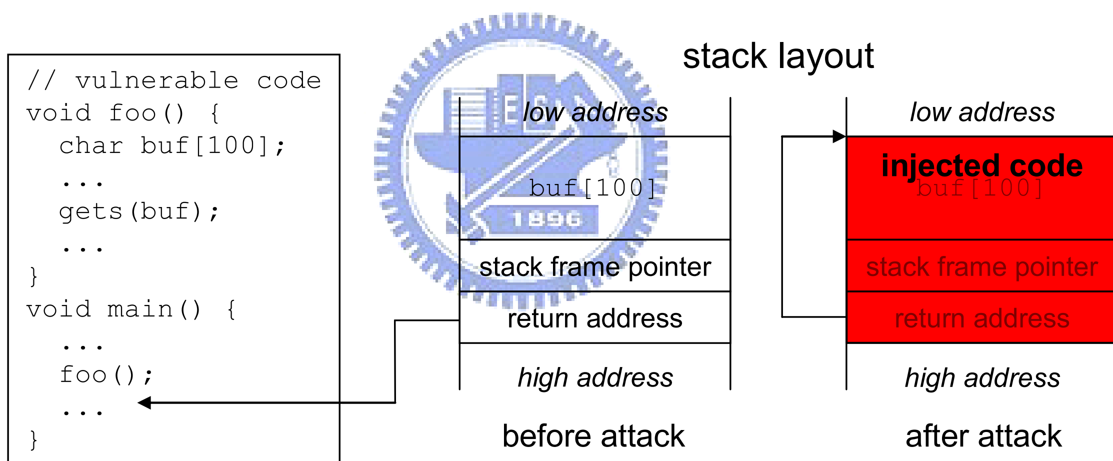


圖 1-1　具有漏洞的程式碼及受到攻擊時記憶體的變化

惡意破壞者會利用溢位存取直接覆蓋掉一些足以控制程式執行流程的重要資料，例如函式指標(function pointer)或回傳位址(return address)，當這些被更改的位址資訊被載入到 CPU 的指令指標暫存器(instruction pointer register, IP)中，CPU 將會去執行任何攻擊者想執行的程式碼，包括具有惡意的程式碼。圖 1-1 顯示了一個有漏洞的程式受到堆疊區破壞攻擊(stack smashing attack) [2]前後的堆疊區資料配置，可以看到回傳位址位於緩衝區 buf 的後方，當輸入的資料長度大於 100 時就會發生溢位，若因此覆蓋到回

傳位址，則在該函式結束時就會跳到被修改過後的回傳位址所指向的位址去執行。

還有一種攻擊藉由間接覆蓋掉程式中某些重要資料，影響程式執行錯誤的行為。例如一個判斷執行權限的變數正好置於可被溢位存取的位址，攻擊者因此得以任意修改權限。

1.2 相關緩衝區溢位防禦工具探討

既然緩衝區溢位漏洞是一個嚴重的問題，許多解決方法已被提出來處理這個問題。

1.2.1 原始程式碼分析技術

這類技術[11][24][34][18][10][9]試著去分析程式碼找出可能漏洞的所在，產生警告告知程式設計師以修改成正確的程式碼。然而目前的研究結果並不佳，經常會產生相當大量不正確的警告。一些研究試著在原來的程式碼分析技術上結合執行期(runtime)存取動作的檢查，例如 CCured[20]或 Cyclone[14]，它將靜態分析階段較難判定的程式碼片段，在執行時期進行複查的動作，如此改善了誤報率且不會造成程式碼太大負擔。

由於這類技術必須要能得到原始程式碼，因此只能應用在程式發展階段，對於現存的一些已部署的應用程式無法立即提供防禦。

1.2.2 執行期溢位偵測技術

這類技術是在編譯前或編譯時插入保護程式碼或是直接修改可執行檔，當執行時所插入的保護程式碼會偵測出溢位存取。目前這種技術依保護的方法可分為下面幾類：

1. 不可執行的區塊：由於緩衝區溢位攻擊通常會在去執行存放在記憶體中的惡意程式碼，透過限制記憶體區塊的執行權限以防止溢位攻擊成功。例如 Non-executable Stack[29]、Non-executable Heap[21]。

2. 間隔：因為緩衝區溢位會通常會連續的存取記憶體，所以可在緩衝區與其他資料或其他緩衝區中插入一段空間，當該空間被存取即可得知溢位。若該空間被存取時馬上能得知稱為強制性間隔(strong separation)，例如 ElectricFence[22]；若該空間被存取一段時間後才能得知稱為被動性間隔(weak separation)。例如 StackGuard[7]。

3. 確保完整性：因為緩衝區溢位會覆寫掉其他資料，所以此技術去檢查其他資料的是否與原先一致以測知溢位是否發生，或是透過加解密資料使其完整性不被破壞。例如 Libverify[3]、StackShield[33]、PointGuard[6]。

4. 邊界檢查：檢查是否超出緩衝區的邊界。例如 BCC(Bounds-Checking C) [16]、CASH(Checking Array Bound Violation Using Segmentation Hardware) [17]、CRED(C Range Error Detector) [25]、Libsafe[3]、J&K(Jones and Kelly checker) [15]。

表 1-3　一些執行期溢位保護機制的比較

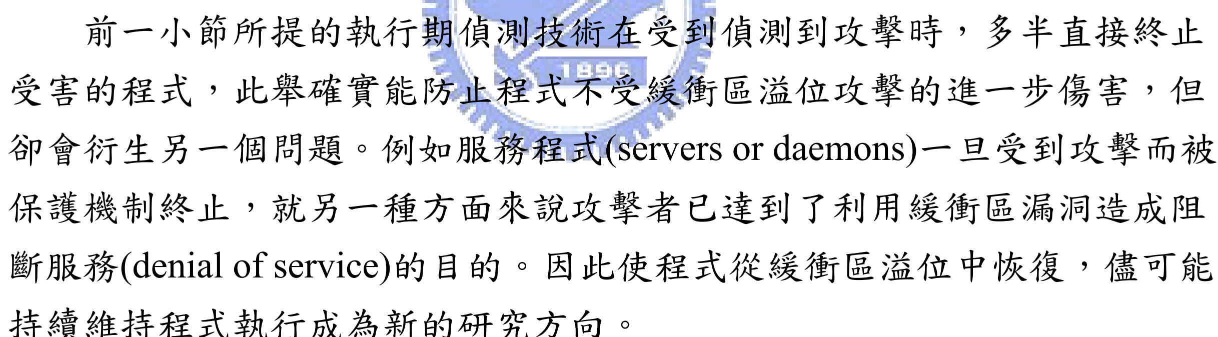
	Targets	Deployment	Overhead	Recovery
Non-executable Stack	Stack buffers	Kernel patch	0	No
StackGuard	Return addresses	Recompile programs	~0	No
ElectricFence	Heap buffers	Runtime instrument	~0	No
PointGuard	All pointers	Recompile programs	0-20%	No
Libverify	Return addresses	Runtime instrument	~0	No
CRED	All buffers	Recompile programs	30%-3000%	No
BMB	All buffers	Recompile programs	30%-3000%	Yes

從表 1-3 所列的內容，我們可以看到不可執行的區塊、間隔與確保完整性這三類技術的效能通常比較好甚至近乎原程式，一個原因是藉助硬體

支援的保護機制；另一個原因是只保護與程式執行相關的資料，例如回傳位址與函式指標，導致其防禦的範圍不夠廣，只能解決部份的緩衝區溢位問題。

第四類根據檢查記憶體存取動作的範圍決定防禦的範圍，但普遍有範圍越廣效能越差的問題。根據麻省理工學院的 Lincoln 實驗室評估[23]，CRED 是目前防禦範圍最廣的偵測工具。它修改了編譯器，當程式每次要存取緩衝區的資料時，插入檢查程式碼，檢查該存取動作是否超出緩衝區的邊界。然而 C/C++並未定義緩衝區長度資訊，故需以某種資料結構維護緩衝區長度資訊。資料結構的操作卻造成了效能的低落。因為溢位存取發生次數的相比於正常的緩衝區存取仍屬於極少數，且執行期偵測機制主要是用以防止隱性的漏洞，為了不知道是否發生的溢位問題，如此犧牲效率的代價太大。

1.2.3 執行期溢位恢復技術



前一小節所提的執行期偵測技術在受到偵測到攻擊時，多半直接終止受害的程式，此舉確實能防止程式不受緩衝區溢位攻擊的進一步傷害，但卻會衍生另一個問題。例如服務程式(servers or daemons)一旦受到攻擊而被保護機制終止，就另一種方面來說攻擊者已達到了利用緩衝區漏洞造成阻斷服務(denial of service)的目的。因此使程式從緩衝區溢位中恢復，儘可能持續維持程式執行成為新的研究方向。

Execution Transaction[28]以類似資料庫處理的方式來恢復溢位問題。這個機制將每個函式視為一個處理單位(transaction)，若該函式中的緩衝區存取發生溢位，則中斷(abort)該函式並且跳回上一層函式繼續執行。但該研究中做了許多的 C/C++語意上的假設，因此實際上是否可行仍待評估。

BMB(Boundless Memory Blocks) [23]是另一個恢復機制。其基本的想法是因為緩衝區有大小限制，所以造成了緩衝區溢位，若緩衝區無限大就不會有緩衝區溢位發生。它以 CRED 為基礎，檢查每次緩衝區存取，若溢位發生，則配置一個新的記憶體區塊，並將溢位存取導向到新的記憶體區塊，

之後再恢復程式執行，如此溢位存取就無法覆蓋到其他記憶體中的資料，而且程式仍能繼續執行。這個技術最大的缺點是繼承了 CRED 的嚴重效能負擔。

1.2.4 資料位址混亂技術(Obfuscation)

由於相同的程式在相同的作業系統上的記憶體配置方式相同並且有一定的規律，通常利用溢位漏洞的攻擊者能夠透過模擬或或計算出需要覆蓋資料的位址。所以有研究[21][5]利用隨機改變資料配置位址來防堵攻擊者達到目的。這類技術配置的原則有三種：1. 隨機改變記憶體區塊起始位址；2. 改變資料配置的高低位址順序；3. 各個區塊間會有任意大小的間隔。

此類技術造成的效能負擔相當低，程式變動也不大，確實可利用相當低的攻擊成功機率來退卻想嘗試攻擊的惡意破壞者。但是這個機率卻無法達到像密碼學所要求的那麼低，有耐心的攻擊者仍可在一段時間後成功的攻擊。

1.3 研究目標與方向

經由前一節的調查，為了要能夠較為快速的解決現今的緩衝區問題，我們以執行期恢復技術為研究目標。恢復技術使用的時機在程式部署之後，為了不造成原程式太大的負擔，效率是首要的考量。

在本篇論文中，試著改善 BMB 恢復機制無法兼顧保護範圍與效率的問題。BMB 效能低落的主因是由於用來偵測溢位的部分是透過 CRED 以軟體方式去檢查每次記憶體存取是否超出所屬緩衝區的邊界。為了要提升效能，利用作業系統與硬體來支援檢查的動作，以事件回報檢查結果(event-driven)是最佳的改進方式。在偵測到溢位後，透過容納溢位的資料讓應用程式能夠從溢位錯誤中恢復，使其不至於因溢位發生而中斷或改變其正常執行。並透過離散配置緩衝區的方式，使每個緩衝區後都能有一塊不小的未配置區域，因而恢復機制能夠以相當有效率的方式進行。本論文以

事件驅動方式偵測為基礎的執行期恢復技術為研究主題所提出的技術稱作BODAR(Buffer Overflow Detection And Recovery)。

1.4 論文大綱

本論文其他章節論述的重點如下：第二章談論設計的主要理念與實作時碰到的問題以及解決方法;第三章驗證 BODAR 的能力並衡量了 BODAR 的效能是否有達到我們所預期的目標；第四章討論 BODAR 在現有系統中所受到的一些影響；第五章總結整篇論文並提出未來可能的改進方向。

第二章 設計與實作

ElectricFence 是一個緩衝區溢位偵測的簡單機制，藉著作業系統與硬體支援的分頁保護機制，當程式呼叫 malloc 配置緩衝區時，位於緩衝區之後且相鄰的分頁將作為保護分頁而不配置。若存取的位址超出緩衝區邊界而位於保護分頁中時，會發現該位址無法對應到實體記憶體空間，作業系統因此發出 SIGSEGV 訊號(signal)。引入 ElectricFence 的程式與 gdb 這類的除錯程式一起執行時，發生溢位的指令將會隨著訊號顯示出來。然而若緩衝區溢位不是連續的，而是以離散、跳躍的方式進行，則溢位跳躍一整個保護的分頁時（通常是 4 KB），這個例外錯誤(exception)將不會被觸發。

J&K 檢查機制去驗證每次的指標運算是否合乎指標所屬的緩衝區邊界，若運算結果超出邊界，該指標將被 ILLEGAL(-2)值取代，任何透過該指標存取記憶體將會造成錯誤。J&K 的問題在於若運算結果為暫存值並沒有實際存取記憶體時，暫存的結果會被修改為-2 導致其後運算錯誤而程式無法正常執行。CRED 為了解決 J&K 問題，透過將錯誤的指標指向一個中繼的物件，該中繼物件會存放運算錯誤的位址，如果錯誤的指標為暫存值，那麼其後的程式能透過中繼物件所存位址運算得到正確值。BMB 以 CRED 偵測機制為基礎加上了恢復機制，當它偵測到溢位時會額外配置一個記憶體空間，透過位址轉換將溢位資料導向該空間。最近的研究認為 CRED 偵測範圍廣且低誤報率，在現存的數種著名的偵測工具中是最佳的。主要的缺點在於不可接受的高執行負擔，某些情形下會有約 30 倍於原程式的落差。

BODAR 主要結合 ElectricFence 的偵測方法與 BMB 的恢復概念並加以改進，認為全部的虛擬位址空間都可被視為防禦溢位存取的柵欄(fence)。每一個配置的緩衝區將以相隔最大距離的方式隨機分配到這些位址空間中。既然大多數虛擬位址是未配置實體記憶體的區段，任何高低溢位(overflow,underflow)與跳躍式溢位都會觸發分頁錯誤。若程式執行上不發生溢位，直覺上不論是程式結果或是執行效能將與原先程式一樣。若溢位存

取，這些未配置的空間將會使得在溢位指令實際執行前先觸發訊號處理函式。

BODAR 的方法簡單但能使應用程式擁有偵測能力與恢復能力。偵測機制像是一個小型的記憶體管理系統，使用 mmap 系統函式離散地在未配置的區段中配置記憶區塊，保護任一個配置出的緩衝區。恢復機制像是作業系統的分頁錯誤處理機制，快速地排除錯誤恢復執行。下面詳細的介紹 BODAR 的保護方法與實作時所碰到的問題。

2.1 緩衝區的組織與配置

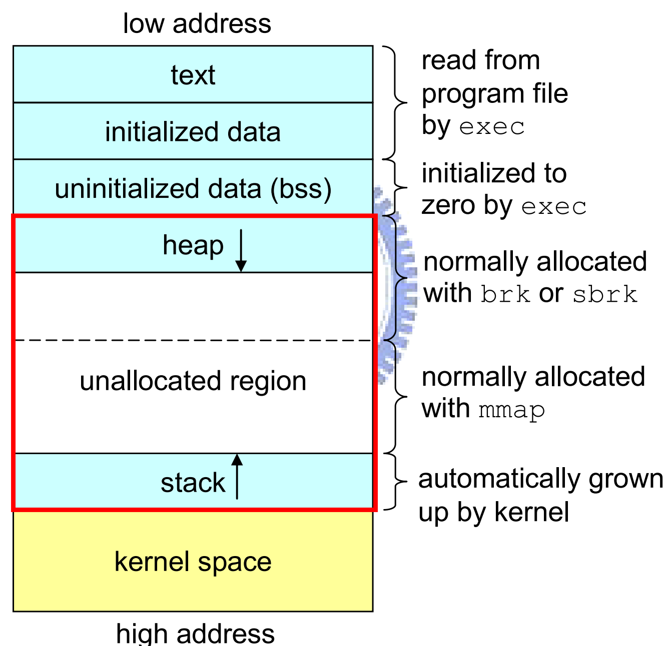


圖 2-1　一個 FreeBSD 程序對虛擬記憶體空間的使用狀況

時下的作業系統提供每個程序(process)分別獨立的虛擬位址空間(virtual address space)。以 FreeBSD 為例，如圖 2-1 所示，在這虛擬位址空間中，堆疊區存在於使用者程序位址空間的高位址區域；而堆積區存在於本文區與靜態資料區（包含已被初始化與未初始化的資料）之後。在堆疊區與堆積區中間為未配置實體記憶體的區域，這塊區域被存取時因為無法對應到實際的記憶體，故會產生區段違反存取訊號(segment violation signal, SIGSEGV)。堆積區一般透過系統函式 brk 與 sbrk 向高位址增長；堆疊區

則由作業系統內部自動配置並向低位址增長。這些增長通常是連續的，即是說，新配置的記憶體區塊將與之前配置的記憶體區塊相鄰。

除此原來的堆積區與堆疊區增長方式外，還可利用系統函式 mmap 直接在未配置實體記憶體區段進行配置。原來的用法是為了映射裝置或檔案到配置的記憶體位址空間中，方便利用記憶體直接對裝置或檔案進行存取。除此之外，mmap 亦被用在建構共享記憶體與跨程序通訊機制。

另一方面 mmap 也可用來配置位址空間給實體記憶體。相比於透過 brk/sbrk 的只能連續配置記憶體的運作方式，mmap 可以指定記憶體區塊配置的起始位址，即是說允許新區塊不需要與任何舊區塊相鄰。因此得以進行非連續地配置記憶體空間的動作。BODAR 利用了這個能力，使緩衝區與緩衝區之間都保有一段未配置實體記憶體的區段，用來阻斷溢位存取到下一個緩衝區。

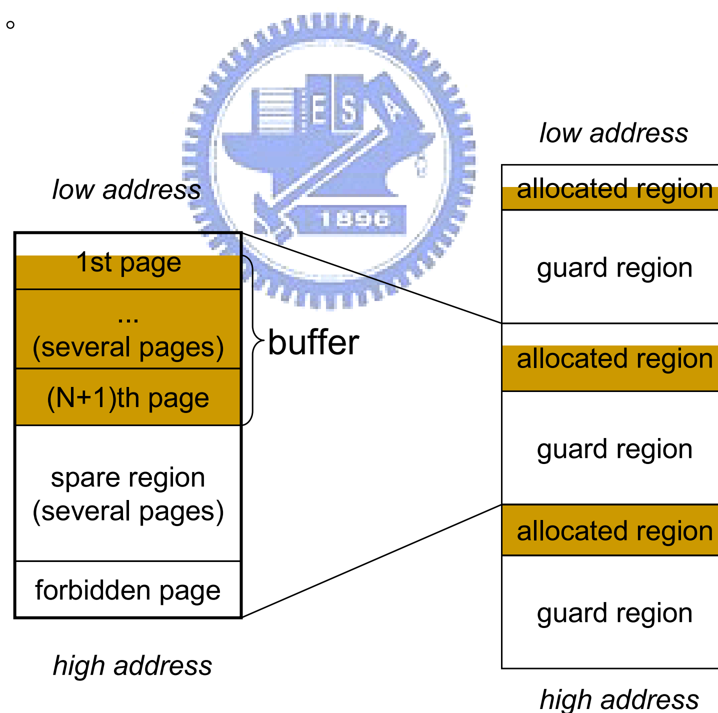


圖 2-2　BODAR 緩衝區的組織方式

為了能正確偵測溢位，緩衝區起始位址須小心調整。假設要配置的緩衝區大小等於 N 個分頁，則起始位址恰好為第一個分頁開頭；若大於 N 個分頁並小於 N+1 個分頁，則起始位址將被往後位移，使得緩衝區的末端能

夠恰好對齊第 N+1 個分頁的末端，見圖 2-2。如此可見超出緩衝區的存取將馬上觸發 SIGSEGV 訊號並立即進行恢復。

BODAR 所配置的緩衝區組織如圖 2-2 所示。除緩衝區佔用的分頁以外，其它均為未配置實體記憶體的區域。由於已配置的緩衝區與未配置的位址空間為交錯排列，為了方便後來的說明，緩衝區其後跟隨的未配置位址空間稱為該緩衝區的守衛區域，其中依處理的任務再分為備用區域和禁用區域。禁用區域大小只佔一個分頁空間，備用區域大小依所屬緩衝區至下一個緩衝區的距離而定，不過最少會配置一個分頁空間。

備用區域在設計時被用來容忍緩衝區溢位資料；而禁用區域被用來限制繼續容忍溢位資料。為了要能盡量提昇容忍的能力，故我們配置位址空間的原則是盡量加大守衛區域的空間。我們採取的方法很簡單，由於緩衝區會分割出許多未配置區段，下一個緩衝區要配置時將從這些未配置區段中選取一個最大的，並配置要求的緩衝區在該區段中間。如圖 2-3，經過 K 次配置後緩衝區 A 之後跟隨的守衛區域是所有守衛區域中最大的，其大小為 M 個分頁。若第 K+1 次將配置一個會佔用 N 個分頁的緩衝區 C，則緩衝區 C 將自緩衝區 A 之後第(M－N)/2+1 的分頁開始配置。

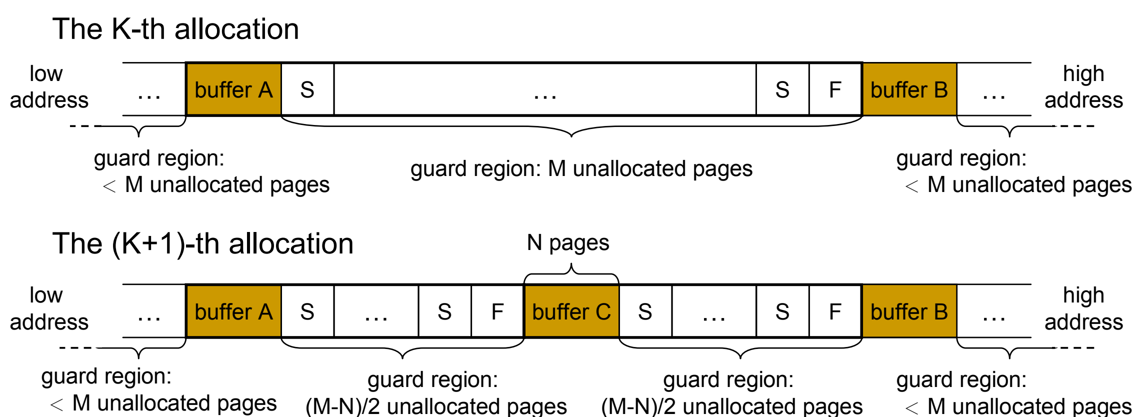


圖 2-3　BODAR 緩衝區的配置方式

但是這種離散配置方法會使未配置區段的大小縮減的很快，造成即使未配置區段空間大小總量仍足以容納要求配置的記憶體大小，卻無法找到適當的未配置區段。為了避免這個問題，我們保留一個仍以原先緊鄰、連

續配置模式的區域，用來當找不到空間時的配置之用，在這區域中仍提供 BODAR 的恢復與保護的機制，但緩衝區的備用區域被固定為只有一個分頁的大小。

原先作業系統管理堆疊區機制是透過建表(table)，表中的項目(entry)存放一個指向資料結構指標，資料結構維護該項目所對應的分頁的資訊，由於是以分頁起始端的位址作為項目索引，故其為相當有效率的一個實作方式。但在 BODAR 設計概念上，因為緩衝區與緩衝區間相隔相當大的未配置區域，若仍套用原先的管理方法，未配置區域中的分頁仍會佔用表中的一個項目，造成表的大小暴增。這種情況可預見在 64 位元的電腦上會更嚴重。我們估計在 64 位元電腦上的位址空間約為 131035 GB，若採用原先方法，則第一次配置於位址空間的中間就會花掉 17175019520 個項目，一個項目需要 8 位元組存放指標，會佔用約 128 GB 的實體記憶體空間。所以 BODAR 改採 AVL 樹來管理緩衝區記憶體配置。以緩衝區為單位取代原先以分頁為單位，每個樹節點存放一個緩衝區資訊，並以該緩衝區起始位址為節點鍵值，如此將能大幅減少管理所佔用的記憶體並有不錯的效能。

2.2 錯誤位址解析

為了能從溢位存取中恢復，首先必須先知道溢位存取是由哪個緩衝區產生的，才能正確對該緩衝區做處理。這點可以利用溢位的位址來判斷。當溢位位址位於該緩衝區後的守衛區域內時，經由 AVL 樹的查詢可馬上判斷是哪個緩衝區發生溢位。

通常廣被使用的 ANSI C 訊號處理函式原型(prototype)除了訊號編號以外並沒提供其他相關的資訊。幸運的是 POSIX 的訊號處理函式原型提供了我們所要的資訊。以下是這兩個處理函式的原型：

```
ANSI C:    void handler(int);
POSIX:     void handler(int, siginfo_t *info, void *uap);
```

FreeBSD 與 Linux 都有支援 POSIX 函式原型。在 POSIX 的函式參數中，info->si_addr 存放著觸發訊號時的指令存取記憶體的位址，而這正是我們所想要的。為了使得作業系統能處理 POSIX 函式原型，須以系統函式 sigaction 替代 signal 註冊，並且開啟 sigaction 第四個參數 sa_flags 的 SA_SIGINFO 旗標。

上面所述的方法已在 FreeBSD 與 Linux 上測試過。兩者除了在第三個參數的資料結構上略有不同外，並不影響到錯誤解析方法的運作。

2.3 溢位恢復處理

引入 BODAR 的程式，在一開始時會向作業系統註冊一個新的 SIGSEGV 處理函式，當溢位發生後，會促使註冊的函式進行恢復動作。假設當時緩衝區 A 的備用區域 SA 有 k 個分頁，依位址由低到高分別為 SA_1 至 SA_k。禁用區域 FA 佔用一個分頁。處理的細節如下：

1. 經由前一節所述之方法得到發生溢位的位址，假設該位址屬於備用區域 SA 中的分頁 SA_x，$x \in \{1,2,3, \ldots ,k\}$。

2. BODAR 將配置實體記憶體給 SA_x 這個分頁。由於處理函式執行完後仍會從發生溢位的指令重新繼續執行，但這次存取的 SA_x 已經被配置了實體記憶體，故不會再產生 SIGSEGV。

3. 若禁用分頁 FA 被存取，目前的 BODAR 將會在處理函式收到訊息後終止程式以避免在 FA 之後的緩衝區資料被溢位竄改。

4. 在 SA 中配置用來存放溢位資料的分頁，在緩衝區 A 被釋放(free) 後將一起被釋放。

5. 為了避免溢位資料浪費太多實體記憶體資源，我們限制緩衝區增長的長度。設該限制為 t 個分頁，表示當未配置的分頁 SA_x($x>t$)被存取時，BODAR 會將自 SA_t 到 SA_{x-1} 的分頁全部釋放，成為未配置狀態。

2.4 緩衝區的轉換

保護機制若是藉修改程式碼引入，通常時間點可分為編譯前、編譯時和編譯後這三種。第一種使用重寫程式碼工具，將保護碼插入到程式中，再將重寫過後的程式碼交由原來的編譯器編譯。第二種修改編譯器，經修改後的編譯器編譯出的可執行檔將含有保護碼。第三種修改編譯出的目的檔(object file)或可執行檔。引入保護機制的時間越早，能得到越多的緩衝區資訊，並越具有可攜性，保護碼也可經過越多的最佳化處理。相反地，時間越晚雖然在程式執行動作上越清楚，最佳化處理得以改由保護機制掌握，但是與其他程式或函式庫的相容性將會降低。

除了上面三種時間點以外，藉由靜態連結(static linking)或動態載入(dynamic loading)亦是一個簡單可行的方法，只要有一個與緩衝區相關的進入點，即可透過該進入點提供保護，而不需要修改原來的程式。下面小節敘述 BODAR 如何提供各種緩衝區保護。

2.4.1 堆積緩衝區的轉換

因為程式可攜性的關係，大部份堆積緩衝區的配置與釋放都是透過 `malloc/realloc/free` 這類 C 標準函式，故我們可以重新實作這類函式，由靜態連結或動態載入修改過後的函式庫達到將新函式取代舊函式的目的。動態載入可透過現今作業系統普遍支援的預載(preloading)特性[3]，藉環境變數 LD_PRELOAD 或定義檔/etc/ld.so.preload 定義可動態載入的函式庫，使引入更為容易與彈性。

2.4.2 堆疊緩衝區的轉換

導入 BODAR 到堆疊緩衝區或靜態緩衝區不是像堆積緩衝區那麼容易。由於堆疊緩衝區的配置僅需改變 CPU 框架指標暫存器(frame pointer register)，程序執行時完全不會呼叫任何配置函式；為了要導入 BODAR 到這類非堆積緩衝區中，我們採用程式碼重寫技術，轉換所有的堆疊緩衝區為堆積緩衝區並且不影響程式的語意。重寫過後再重新經由編譯器編譯，應用程式將可以使用我們的恢復技術在堆疊緩衝區溢位上。

我們用來做轉換的方法主要是根據 Dahn 與 Mancoridis 的研究[8]，這篇論文為了要避免堆疊緩衝區溢位攻擊，提出轉換所有的堆疊緩衝區為堆積緩衝區的方法。他們的方法中使用了 TXL[32]去處理程式碼重寫。TXL 是一個混合規則導向(rule-based)與函式導向(functional)的程式語言，特別適合用在程式碼的重寫。為了要能執行 C 程式的重寫，我們需要 C 的 TXL 文法定義和規則說明如何重寫程式碼。前者可在 TXL 的網站上找到，但後者就只能自己定義了。另外我們也利用 Perl 編寫能夠自動做重寫動作的工具，使得 BODAR 在應用時能更方便。

Original code

```
void foo() {
  int buf[64];
  bzero(buf,sizeof(buf));
  ...
}
```

Modified code

```
void foo() {
  int *buf=(int*)malloc(64*sizeof(int));
  bzero(buf,64*sizeof(int));
  ...
  free(buf);
}
```

圖 2-4　堆疊緩衝區修改前後的程式碼

圖 2-4 顯示如何轉變堆疊緩衝區為堆積緩衝區。在原來的程式碼中，`buf` 是在一個函式 `foo` 中的陣列即是堆疊緩衝區。在重寫過的程式中，`buf` 被修改成一個指標，並且以 `malloc` 做初始化，其大小將同於原先的陣列大小。要注意的是原來的 `buf` 並不需要有回收的程式碼，因為在函式回傳時將自動釋放該緩衝區。但相反的，在修改過的程式碼中，緩衝區並不會自動被釋放，因此我們必須在函式回傳前加入釋放的程式碼，確保記憶體使用正確。

在 C 語言中的 sizeof 敘述必須小心處理，原來程式碼中的 sizeof(buf) 會傳回陣列的位元組長度 32*4，但轉換過後 buf 變為一個指標，sizeof(buf) 只會回傳指標的位元組長度 4，故我們必須以呼叫 malloc 時傳入的大小取代原來的 sizeof(buf) 敘述。

2.4.3 靜態緩衝區的轉換

靜態緩衝區的配置是在編譯階段就決定了配置的位址與空間，載入時實際將空間配置給緩衝區。為了要將靜態緩衝區轉換為堆積緩衝區，我們將靜態緩衝區宣告全部改為指標，在程式開始的時候才進行配置。

根據 ELF(Executable Language Format)的定義，可執行檔中的建構區段 CTOR 與解構區段 DTOR 存放許多函式指標。程式開始時會先執行去 CTOR 中的函式，之後才會呼叫標準程式進入點 main，當程式結束後會去呼叫 DTOR 中的函式。實作上可透過 GCC[12]所定義的建構解構函式來插入函式指標至 CTOR/DTOR 區段。以下為建構解構函式的宣告原型：

建構函式原型：

```
void ctor() __attribute__ ((constructor));
```

解構函式原型：

```
void dtor() __attribute__ ((destructor));
```

經由建構解構函式，我們可以將所有的靜態緩衝區改宣告為指標，而在建構函式中才透過 malloc 配置 BODAR 保護的緩衝區，最後透過解構函式將這些緩衝區釋放(free)。此外，在程式中出現的 sizeof 以前一小節所述相同方式處理。圖 2-5 顯示如何轉變靜態緩衝區為堆積緩衝區。

Original code

```
int buf[64];
```

Modified code

```
int *buf;
void bodar_ctor() __attribute__ ((constructor));
void bodar_ctor() {
  buf=(int*)malloc(64*sizeof(int));
}
void bodar_dtor() __attribute__ ((destructor));
void bodar_dtor() {
  free(buf);
}
```

圖 2-5　靜態緩衝區修改前後的程式碼

第三章 實驗與分析

在這章中我們驗證 BODAR 的能力並衡量其效能。我們選取了現今數種開放程式碼的應用程式作為測試套件，這些應用程式雖然廣為使用卻都有已知的緩衝區溢位漏洞。所有的測試均在 FreeBSD 5.4 上執行。實驗中共使用了兩種不同中央處理器的主機，32 位元採用 AMD Athlon XP 2000+中央處理器；64 位元採用 AMD Athlon XP 3000+中央處理器。每台主機均配備 1 GB 隨機存取記憶體(random access memory, RAM)、80 GB 硬碟和 100 Mb 乙太網路卡。這些主機全部被連接到與外界隔離的內部網路中，以避免實驗結果受到外界影響。

3.1 偵測能力

我們用來實驗的 BODAR 保護機制原型並沒有對所有的緩衝區作保護。目前無法提供保護的緩衝區如下所述：

1. 目前 BODAR 並沒有提供 `struct` 與 `class` 中緩衝區和多維緩衝區陣列的保護。因為若將第二章所述之方法應用其上，會大大改變這些資料結構的語意，目前並沒有適當的解決方法，故沒有實作在用來實驗的 BODAR 原型中。希望未來能提供更完善的轉換程式來處理。

2. 我們認為相當多數的已報告緩衝區溢位漏洞並非發生在靜態緩衝區，雖然前文提出了靜態緩衝區的轉換方法，但我們並沒有實作到實驗的 BODAR 原型中，若未來趨勢顯示需要對靜態緩衝區作保護，仍能實作到 BODAR 中。

除了上面所列的緩衝區外，我們挑選了數個均有已知漏洞的應用程式，引入 BODAR 的保護機制來測試，結果都能夠正確的測知緩衝區溢位的發生。我們測試的程式如表 3-1。

1896

表 3-1 一些具有緩衝區溢位漏洞的應用程式

Applcation	Cause
apache-1.3.34+mod_mylo-0.2.1	stack overflow
exim-4.42	stack overflow with strcpy
gawk-3.1.0	stack overflow with strcpy
html2hdml-1.0.3	stack overflow
libpng-1.2.5	stack overflow
o3read-0.0.3	stack overflow
pgp4pine-1.7.6	stack overflow
ringtonetools-2.22	stack overflow
thttpd-2.23b1	off-by-one stack overflow
unalz-0.31	stack overflow with strcat
vb2c-March-1999	stack overflow
vilistextum-2.6.6	stack overflow
xlreader-0.9.0	stack overflow with strcat

3.2 恢復能力

我們透過遠端服務程式來驗證恢復能力的重要性。實驗設計分為攻擊者與受害者，扮演受害者的主機上安裝有漏洞的服務程式，而攻擊者則針對該漏洞以一定的頻率送出造成溢位的服務要求。我們調整這些溢位攻擊只會使受害者主機上的服務程式不正常終止(crash)。

我們使用 Webstone[19]來量測受害者主機上的服務程式的效能，並觀察攻擊頻率與效能間的影響。每次的實驗均測試三分鐘，透過兩台主機上各十個程序對受害者主機上的服務程式送出服務要求，藉由回應衡量效能。

我們選取兩種不同實作方式的網頁服務程式來測試。一種是以多程序同步模式(multi-processes concurrent model)設計的 Apache[30]。一種為單一程序同步模式(single-processes concurrent model)的設計的 thttpd[1]。

3.2.1 Apache

Apache 是個相當知名的網頁服務程式，但也被發現有不少漏洞。本論文中所用來測試的是 Apache-1.3.34 與其附加模組 mod_mylo-0.2.1。mod_mylo 是一個把網頁存取紀錄存放在 MySQL 資料庫中的模組，要在

Apache 中使用 mod_mylo 必須在安裝 Apache 時把動態分享物件(dynamic shared object, DSO)的選項開啟。

因 Apache 在設計上採用多程序同步模式的方法，一開始會預先開啟許多子程序置於程序儲備區(process pool)，當接收到來自網路連線的網頁存取要求時，會從儲備區中選取一個子程序來服務該連線，使用完後放回儲備區以給下一個連線使用。在這實驗中 Apache 的子程序儲備區，被設定為初始 10 個程序，最大 150 個程序。

我們主要利用 mod_mylo 模組上面發現的堆疊緩衝區溢位漏洞[26]來進行攻擊，這個漏洞是由於程式設計師拷貝字串時填入的長度大小錯誤所致。雖然這個漏洞可被用於侵入系統，但是在這裡的實驗中，我們僅僅破壞 Apache 的子程序使其無法再提供服務。

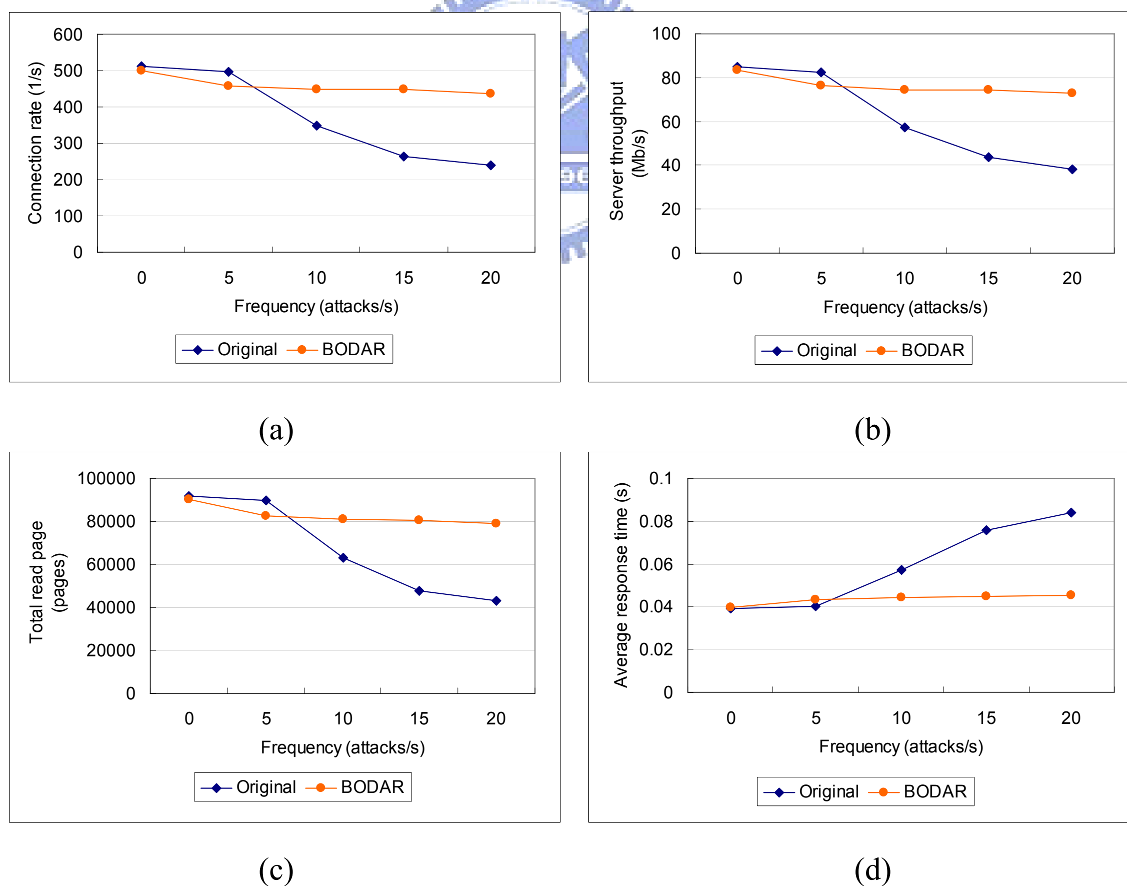


圖 3-1　Apache 與 Apache-BODAR 受到攻擊時服務效能的比較。(a)連線成功率，(b)輸出量，(c)實際存取網頁數目，(d)平均回應時間。

從圖 3-1 我們可以看到未受到保護的 Apache 的表現，隨著越來越頻繁的攻擊，連線成功的比率、輸出量、實際存取網頁數目都呈現快速下滑的趨勢，分別下滑了 53%、55%、53%，而平均回應時間則越來越久，上升了 114%，這些結果都足以顯示其提供服務的能力越來越低落；而 Apache-BODAR 四項服務指標的變化在 12%到 14%內，這是由於恢復機制的額外負擔造成的，但可看到只在無攻擊到有攻擊時影響較大，之後仍能持續穩定的提供服務。

3.2.2 thttpd

thttpd 是一個小型且快速的網頁服務程式。他與 Apache 主要的差別在於是以事件驅動模式設計並實作。在這種模式中並不會產生子程序專門服務一個連線。而是由主程序處理全部的連線，當客戶端(client)透過連線送出要求後會觸發事件，主程序再處理針對該事件做對應的處理。

這裡用來實驗的漏洞是一個 off-by-one 的堆疊區溢位漏洞[27]。造成 off-by-one 漏洞一開始是因 C/C++處理字串時結尾需要補上一個 NULL 字元，字串緩衝區的大小需要比字串長度要多 1，若程式設計師仍以原來字串長度做為緩衝區大小就會造成溢位。後來 off-by-one 漏洞被廣義地認為程式設計師寫程式時仍有考慮到緩衝區問題，只是在處理時計算錯誤，造成極為有限的溢位。雖然溢位有限但發生在 thttpd 的這個漏洞仍會促使 thttpd 主程序受到破壞而終止。

如同前一節所進行的實驗。與 Apache 較不一樣的地方是，由於受到攻擊後 thttpd 主程式被終止後無法再接受任何連線，需要透過外部的監控程序重新啟動，而監控程序被設定為每秒檢查一次主程序是否存在，所以受到攻擊後不能馬上重新啟動，因而影響會比只需重新生出子程序的 Apache 嚴重。我們在 thttpd 實驗中所用的攻擊頻率並不像在 Apache 實驗時那麼頻繁。但為了與之前實驗數據單位相同，實驗數據中會有每秒 0.5 次攻擊這類資料出現，換句話說即是 2 秒進行一次攻擊。

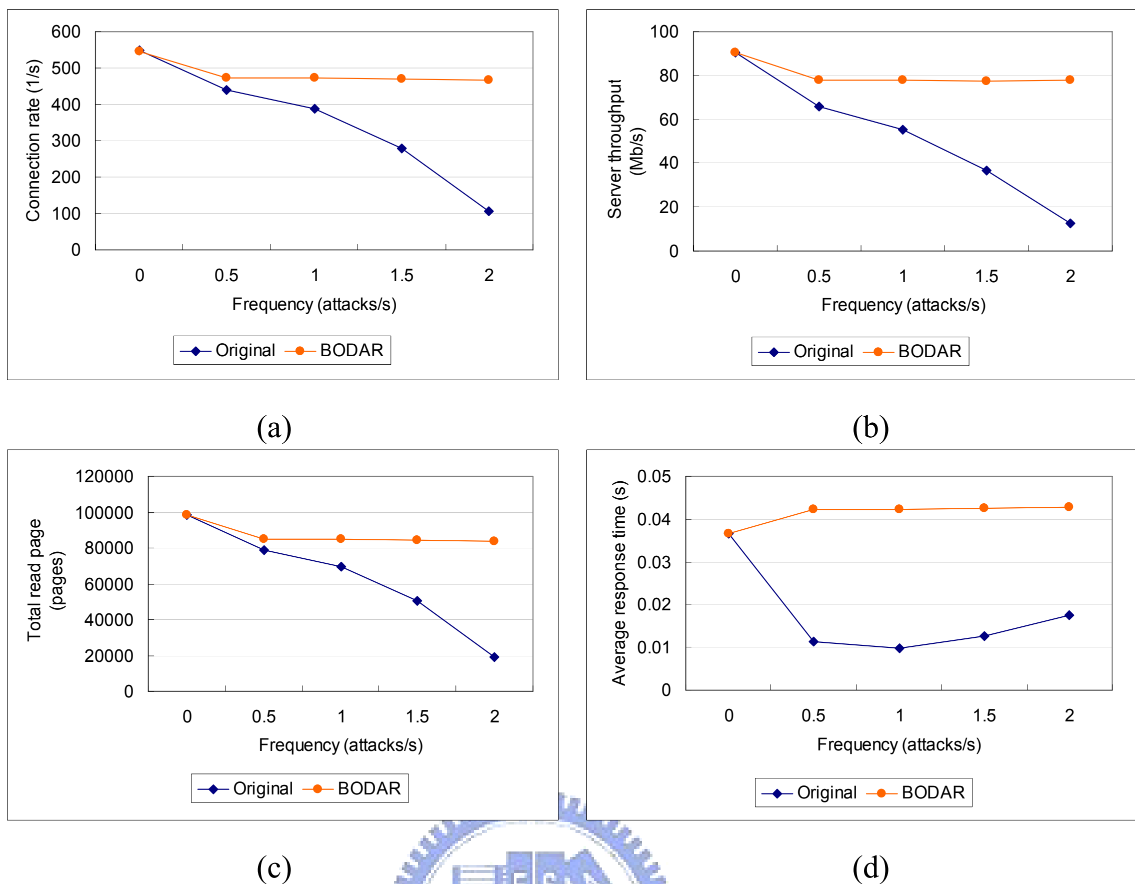


圖 3-2 thttpd 與 thttpd-BODAR 受到攻擊時服務效能的比較。(a)連線成功率，(b)輸出量，(c)實際存取網頁數目，(d)平均回應時間。

圖 3-2 顯示了實驗結果。前三項結果顯示 thttpd-BODAR 因為不需要重新啟動，故依然有效的維持住提供服務的能力。然而，第四項回應時間的結果卻與預期相反，原因是 Webstone 在計算回應時間時只採計成功連線後到連線中斷時的時間，但這其中卻包含了程式被攻擊後所導致的連線中斷，並不是完整傳送完畢的時間，加上量測的過程中，大檔案因回應時間過久通常會被中斷而不採計，計算的多為回應時間快速的小檔案，故會出現原來版本的回應時間比 thttpd-BODAR 還要快的情形。

3.3 效能評估

因為 C/C++不嚴謹、自由的程式特性，程式設計師在發展程式時相對不容易除錯。通常在程式初步完成後，將程式當做黑箱，透過外部任意的輸入以檢查程式中是否有問題。但是這種檢查並不完善，仍會造成許多隱

含的漏洞不被發現。衡量效能的目的在於我們需要一個能夠與在應用程式部署之後，仍能伴隨著保護的機制。它能夠在盡量不影響程式正常執行效能的情況下，具有緩衝區溢位漏洞偵測與恢復的能力。

在這節中量測 BODAR 保護的程式與正常程式相比所增加的效能負擔。原先實驗預定比較的技術為 BMB，但因為無法取得 BMB 的程式碼，所以改以 CRED 為做為衡量的對象。因為 BMB 根植於 CRED，也繼承了絕大部分 CRED 的效能負擔，故能藉 CRED 評估 BMB 的效能。下面測試的應用程式均編譯為三種版本，分別為原始版本，BODAR 版本和 CRED 版本。我們分別實驗兩種不同類型的程式來評估。

3.3.1 重視輸出輸入(I/O-bound)的應用程式

我們衡量 Apache 與 thttpd。其中 CRED 無法正確編譯 thttpd，故 thttpd 不比較 CRED 版本。我們仍以 Webstone 巨觀地量測這兩個網路服務程式正常狀況下的效能。

表 3-2 Apache 與 thttpd 在無攻擊時，服務效能的比較

	apache			thttpd	
	Original	CRED	BODAR	Original	BODAR
Server connection rate (connections/sec.)	558.31	377.78	521.26	547.20	546.50
Ratio to Original	100%	-32.34%	-6.64%	100%	-0.13%
Server throughput (Mb/sec.)	92.24	63.20	86.58	90.51	90.51
Ratio to Original	100%	-31.48%	-6.14%	100%	-0.00%
Total number of pages read (pages)	100496.33	67999.33	93826.33	98496.00	98370.67
Ratio to Original	100%	-32.34%	-6.64%	100%	-0.13%
Average response time (sec.)	0.035778	0.052848	0.038323	0.036508	0.03655
Ratio to Original (less is better)	100%	+47.71%	+7.11%	100%	+0.13%
Maximum virtual data size (KB)	3296	2684	3504	9852	11256
Ratio to Original (less is better)	100%	-18.57%	+6.31%	100%	+14.25%

如表 3-2 顯示，Apache 與 thttpd 的 BODAR 版本在四項服務性指標中都勝過 CRED 版本，並與原來版本的差異量都在 8%以內，仍在可接受的額外負擔之內，這顯示我們的確達到了能提供緩衝區溢位問題的解決方案在部署之後的 Apache 與 thttpd 中。此外，由於 BODAR 配置一個緩衝區至少

會花費一個分頁以維持偵測機制，此舉顯然會浪費實體記憶體空間。所以我們的實驗也衡量了實際的記憶體資源浪費情形。結果顯示 Apache-BODAR 多花費的記憶體空間在 7%以內；thttpd-BODAR 則在 15%以內。 這顯示了實際上 Apache 與 thttpd 並沒有同時間配置太多的緩衝區。

表 3-3　Apache 與 thttpd 的守衛區域大小比較

	Guard region size (KB)			Memory
	Maximum	Average	Minimum	Utilization
Apache	29491	23240	14732	0.0239%
thttpd	7368	4954	3672	0.1787%

我們也量測了在程式執行過程中，BODAR 版本的每個緩衝區的間隔距離，即守衛區域的大小。表 3-3 顯示 thttpd-BODAR 平均間隔約 4954 KB；Apache-BODAR 由於會產生子程序，每個子程序的虛擬位址空間又分別獨立，故會有較大的間隔距離，平均約 23240 KB。

3.3.2 重視 CPU 運算(CPU-bound)的應用程式

在這小節內我們挑選了 gnupg-1.4.1 與 gawk-3.1.0 來評量。前者主要用來產生電子信件的訊息文摘(message digest)和簽章(signature)，其執行的時間隨電子信件的大小增加成一定比例增長。後者為一個簡單的腳本程式直譯器(script interpreter)，執行的時間與腳本程式的大小與複雜度有關。由於這兩個實驗都需要讀入檔案，我們以執行時間為橫軸與記憶體使用量（不包含程式本文所佔的空間）為縱軸，觀察實驗數據對不同大小檔案變化的趨勢。

gnupg 的實驗中將 gnupg 三種版本對六種大小分別為 512 KB、1024 KB、1536 KB、2048 KB、2560 KB 和 3072 KB 的檔案做密鑰大小為 4096 位元的 RSA 加密，之後再使用 gnupg 對前一個實驗後加密過的檔案做解密。我們也量測了 BODAR 在 64 位元電腦上的實驗結果。

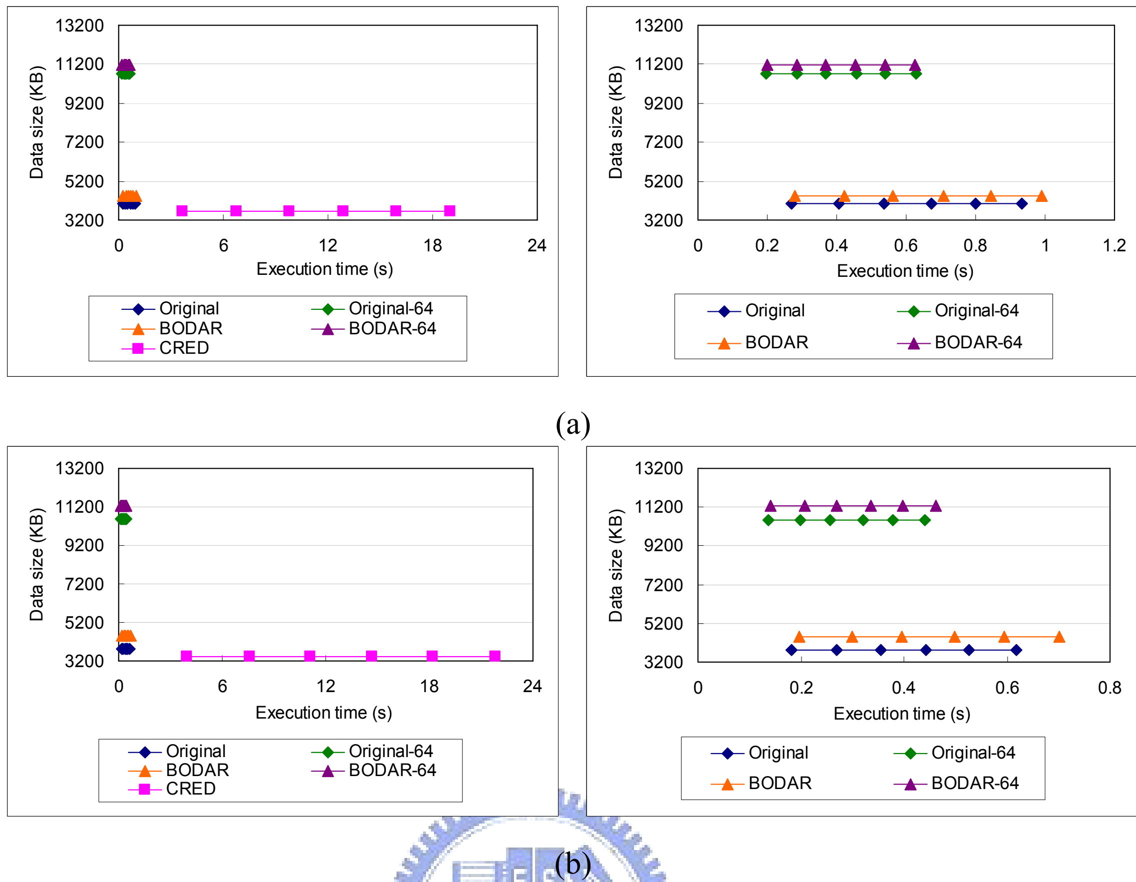


圖 3-3　gnupg 的實驗結果。(a)不同版本的 gnupg 用 RSA 加密六個檔案的結果，(b)不同版本的 gnupg 用 RSA 解密六個檔案的結果。以上右圖均為左圖移除 CRED 版本後的結果。

圖 3-3 可看到執行時間與檔案大小成等量增長，gnupg-Original 的加密與解密執行時間增加量上分別是 0.13±0.006 秒與 0.09±0.006 秒，而 gnupg-CRED 的增加量則是 3.08±0.1 秒與 3.57±0.09 秒，為原先的 23 倍與 41 倍，不難想見 CRED 機制對於程式的效能影響相當大。反觀 gnupg-BODAR 的增加量是 0.14±0.005 秒與 0.10±0.008 秒，為原來的 1.07 倍與 1.16 倍，且在 64 位元電腦上這個倍率會更小，受 BODAR 保護的 gnupg 幾乎不會造成太大的效能負擔。

此外，圖中可見到輸入檔案大小與記憶體使用量無關，在 32 位元電腦上，gnupg-BODAR 在記憶體空間使用量上，加密與解密與原來版本比較分別多了 440 KB 與 700 KB，而在 64 位元電腦上則多了 440 KB 與 712 KB，

雖然在 64 位元的電腦上 gnupg-Original 的記憶體使用量由於基本型態大小改變的關係而增加，但是 BODAR 對記憶體的使用量卻不會因此而增長。

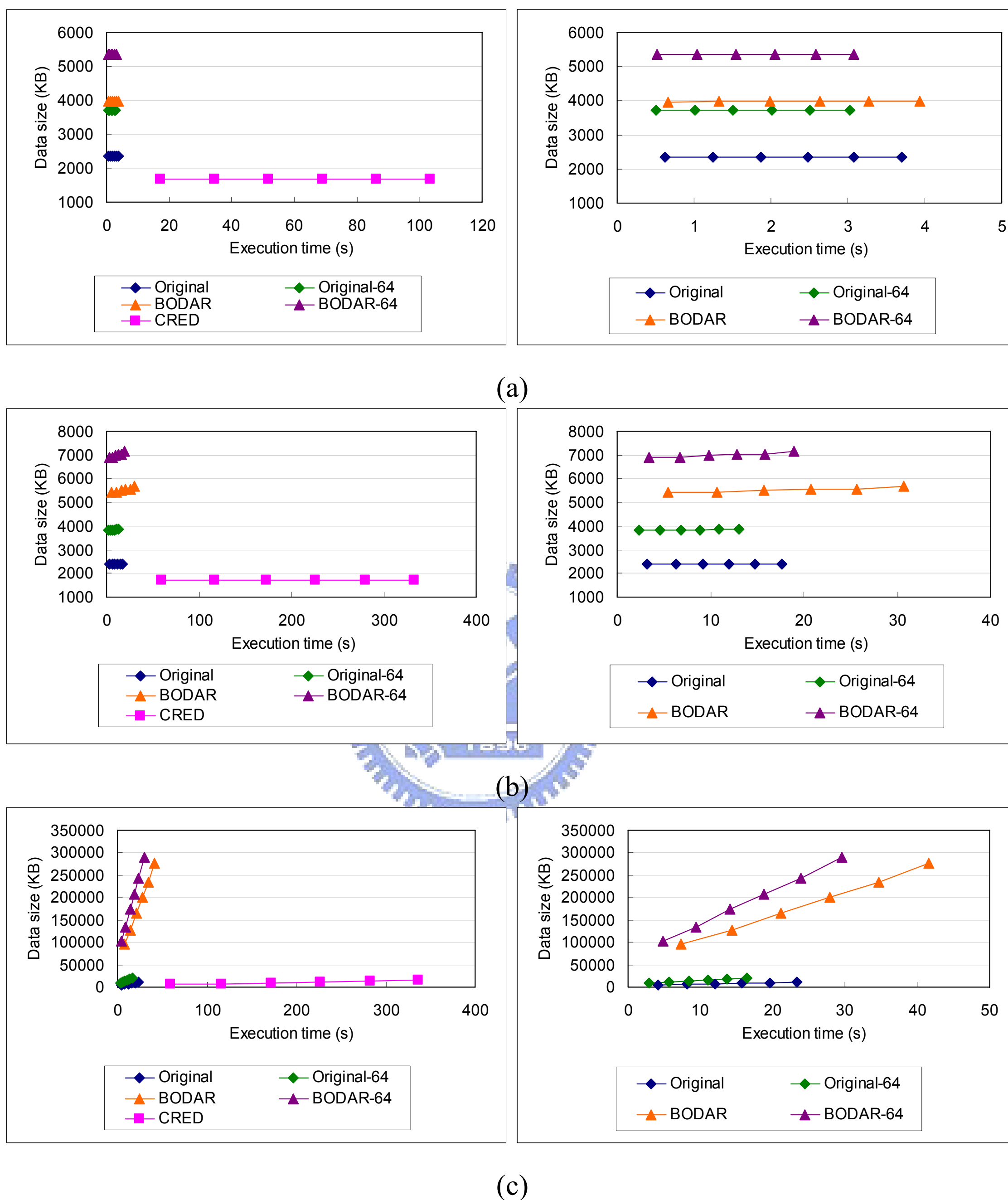


圖 3-4　gawk 的實驗結果。(a) gawk 執行 A 腳本程式，(b) gawk 執行 P 腳本程式，(c) gawk 執行 S 腳本程式。以上右圖均為左圖移除 CRED 版本後的結果。

在 gawk 的實驗方面，我們設計讓 gawk 執行三支不同的的腳本程式，access-time.awk、proxy_stat.gawk 和 scalar.awk（以下分別稱 A、P、S 腳本

程式），這三支腳本程式都是用來處理 squid 代理伺服程式(proxy)所產生出的記錄檔，A 腳本程式的大小與複雜度最小，S 腳本程式最大。實驗結果如圖 3-4。gawk-BODAR 在 A 腳本程式中的效能表現相當良好，只多增加了約 7%的效能負擔。但是隨著腳本程式越來越大且越來越複雜，gawk 需要配置越多的緩衝區，因此效能會負擔越來越高。即是如此 gawk-BODAR 執行 P 與 S 腳本程式時增加的效能負擔約為 80%，相比於 gawk-CRED 的 14 倍至 28 倍，在效能方面 BODAR 版本亦擁有不錯的表現。

在記憶體使用量方面，直譯器需要剖析(parse)腳本程式，經常會進行字串切割並且產生相當多的小型的緩衝區來容納切割的字符(token)。但是 BODAR 在設計上透過硬體分頁保護機制，配置緩衝區的最小單位為一個分頁。當程式越大越複雜將會使用比以往更多的空間。此外，記錄檔大小亦會影響 gawk 配置緩衝區的數目而使問題更嚴重，實驗結果中除了 A 腳本程式對此幾乎無影響外，P、S 腳本程式的記憶體使用量均會隨者紀錄檔大小增加而增加，尤以 S 腳本程式更為明顯。gawk-BODAR 在實驗中最糟糕的情況下將花費 271 MB 的虛擬記憶體使用量，這是原來程式的 25 倍

第四章 討論與改進

4.1 可使用的虛擬位址空間

既然 BODAR 需要盡量最大的間隔以便用來容納可能的溢位資料，因此我們需要找出實際上可使用的虛擬位址空間有多大。圖 4-1 顯示一個受到 BODAR 保護的 FreeBSD 程序在 IA-32 與 AMD-64 電腦中的虛擬記憶體空間的分配情況。依位址低到高順序分別為 BSS 區域、原堆積保留區(original heap)、mmap 保留區(reserved mmap)、新堆積保留區(new heap)、和堆疊保留區(stack)。這裡保留區的意思表示其不一定全部被配置，使用或配置的方法依保留區不同而定。BSS 保留區與堆疊保留區與原來的大小及配置方法不變。新堆積保留區則為採用 BODAR 配置方法的區域。

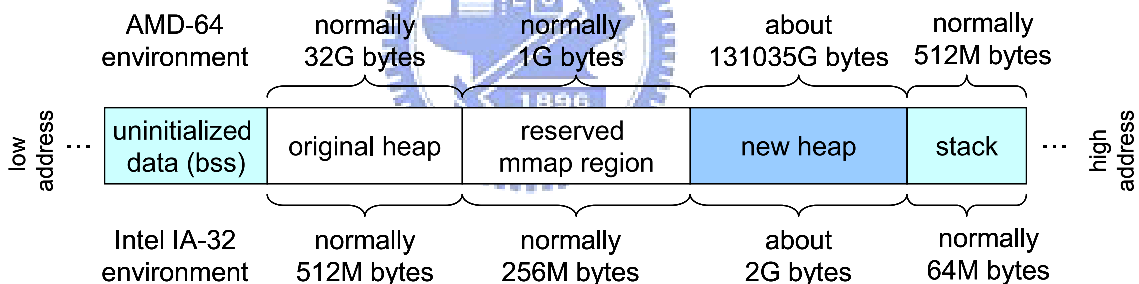


圖 4-1　BODAR 在 IA-32 與 AMD-64 中虛擬記憶體空間使用情況

極少數的情況下應用程式中也會不經由 malloc 配置記憶體空間，而經由 mmap 這個系統函式，例如映射虛擬記憶體區塊到檔案，故我們設計了 mmap 保留區給予系統分配的空間。一般使用 mmap 通常不會指定記憶體區塊的開始位址，故當呼叫此型函式時會以連續的方式自保留區開頭開始分配記憶體區塊。目前我們並沒有針對若指定了起始位址的 mmap 做處理，不過在我們的經驗中應用程式幾乎不會有這樣使用的情況（因為可攜性的關係）。

mmap 保留區的低邊界(lower bound)在程式開始時呼叫未指定起始位址的 mmap 而得到的。低邊界加上保留區設定的大小後，我們可以得到 mmap 保留區的高邊界(upper bound)，即新堆積保留區的低邊界。而新堆積保留區的高邊界則是計算堆疊保留區的低邊界而得的。我們計算堆疊保留區低邊界的步驟如下：

1. 既然環境變數區緊鄰著堆疊保留區，表示我們可以檢查所有環境變數的位址，最低的位址即為堆疊保留區的高邊界。

2. 透過 getrlimit 系統函式得到堆疊保留區在作業系統中設定的最大大小。

3. 將堆疊保留區高邊界減去堆疊保留區大小，可得到堆疊保留區的低邊界。

從上面的計算可得到在 IA-32 的架構下新堆積保留區可以使用大約 1.8 GB 的虛擬位址空間，既然 BODAR 至少需要 3 個分頁來組織緩衝區，故在最糟的情況下，BODAR 保護的程式只能使用總量約 614 MB 的位址空間，我們在之前實驗最糟糕的情況會花費 271 MB，我們相信這在一般的程式已夠使用。另外就正在推廣的 64 位元 CPU 來看。我們可以用上述同樣的方法得到新堆積保留區的大小約為 131035 GB，實際使用空間為 43678 GB，這樣的大小應該能完全滿足現存所有程式的需求。

4.2 作業系統的記憶體分頁管理機制對 BODAR 的影響

由於記憶體管理與作業系統和硬體緊密相關，而 BODAR 一改過去的記憶體配置方式，故在這節中我們討論一些作業系統設計上對 BODAR 效能所造成影響的因素。

4.2.1 Page Reclaim 對效能的影響

由於虛擬位址空間大小通常大於實體位址大小，再加上每個程序都有自己的虛擬位址空間，所以虛擬位址空間無法一對一的分配到實體位址空間中，所以需透過一個轉換與置換機制，以保證每個程序都能正常使用所有的虛擬位址。

當程序想要存取某個虛擬位址時，若該位址所屬的虛擬分頁沒有對應實體分頁，作業系統會試著從實體記憶體中選取未被配置出的實體分頁並分配給該程序，成功的話由於該分頁被重新配置，故稱做 page reclaim；若失敗則表示記憶體中無法找到未被配置出的實體分頁，發生 page fault，則作業系統必須將部分的分頁置換到硬碟中，以空出實體記憶體的空間。

比較 BODAR 與原來的記憶體分配方式。原來的 `malloc` 函式中以 first-fit 原則將一個分頁中會存放許多小型的緩衝區，即是說要配置好幾個緩衝區才會發生一次 page reclaim。但在 BODAR 中，如之前提過的，一個緩衝區不論大小都至少會需要配置一個實體分頁，每次新配置一個實體分頁都是一次 page reclaim。故應用程式中配置的小型緩衝區多時，page reclaim 會對受 BODAR 保護的程式的效能造成不小影響。

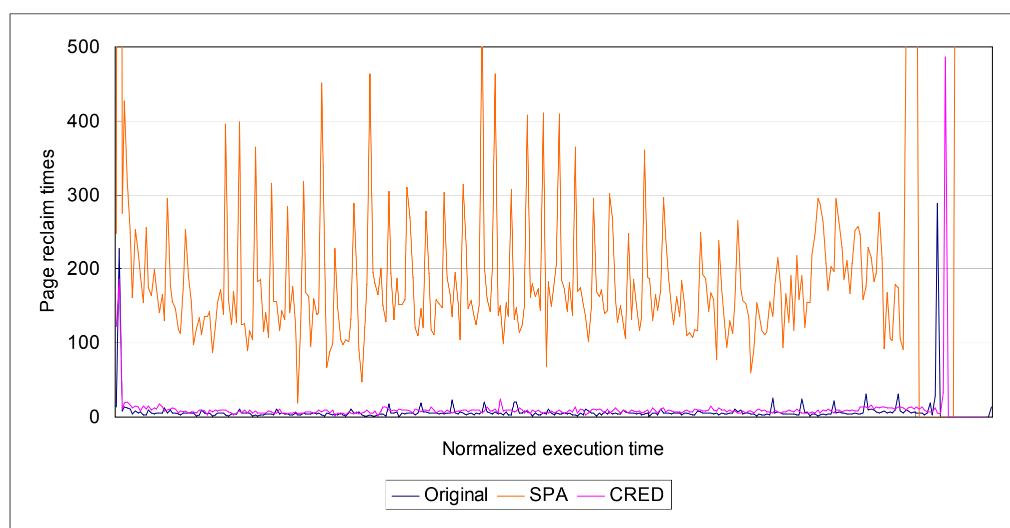


圖 4-2　不同版本 gawk 在執行過程中的 page reclaim 次數統計

圖 4-2 顯示三種不同版本的 gawk 執行 S 腳本程式對 30 萬行紀錄檔統計的結果。縱軸為 page reclaim 的次數，橫軸為程式的正規化執行時間(normalized execution time)。可見到 CRED 與原來版本的 page reclaim 次數相當低，而 BODAR 版本不斷的發生 page reclaim。總計 gawk-BODAR 的 page reclaim 次數約為 68780 次，而程式約配置了 67325 個分頁，故我們可以合理推測幾乎每次配置緩衝區時就會發生 page reclaim。

受 BODAR 保護的程式亦有可能發生 page fault，當實體記憶體不足以提供程式的需求時就會發生，因為它會進行硬碟存取，這對受 BODAR 保護的程式的效能影響將更為嚴重。不過在我們的實驗中並沒有發生此一狀況。

4.2.2 TLB(Translation Look-aside Buffer)對效能的影響

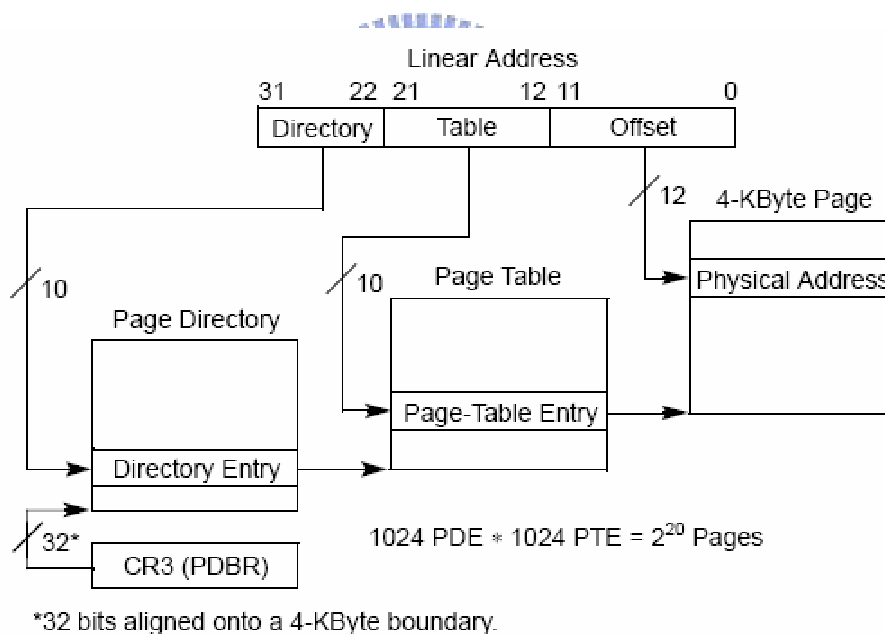


圖 4-3　虛擬位址對應實體位址的轉換機制

資料來源：Intel Architecture Software Developer's Manual vol.3

圖 4-3 顯示在 IA-32 架構 CPU 中作業系統的虛擬位址轉換機制，通常為兩階層分頁模式，第一層稱為分頁目錄(page directory)，第二層稱為分頁表。兩階層的目的是避免把全部的分頁表連續的放在記憶體中而浪費記憶體。當虛擬記憶體位址要轉換為實體記憶體位址時，先從分頁目錄中得到

所屬的分頁表位址，再向分頁表查詢所屬分頁，最後加上位移量。為了要加速此一過程，通常硬體會提供關聯式記憶體 TLB 做為快取，把最近使用的分頁目錄和分頁表的項目存放在其中。

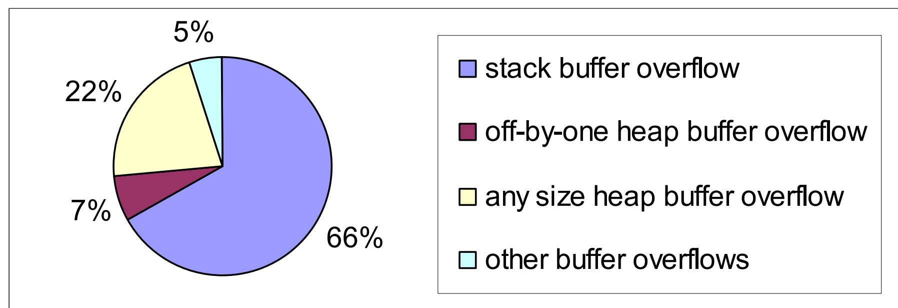
當緩衝區以原來連續且緊密的方式配置時，小於 2048 的緩衝區通常以 chunk 的方式分配到各個分頁中，而 1024 個分頁才會使用到下一個分頁目錄項目，所以分頁目錄與分頁表中使用的項目會很少並且較為頻繁，較能長時間置於 TLB 中而加速位址轉換。相反的 BODAR 卻以離散的方式配置，一開始的 1024 個緩衝區幾乎都位於不同的分頁目錄上，分頁表的情形也一樣，造成分頁目錄與分頁表中使用的項目增加許多，越多的項目會使得快取的效果降低。結果就是只能靠作業系統作位址轉換，降低了程式執行的效能。

4.3 減低記憶體空間浪費的折衷方法

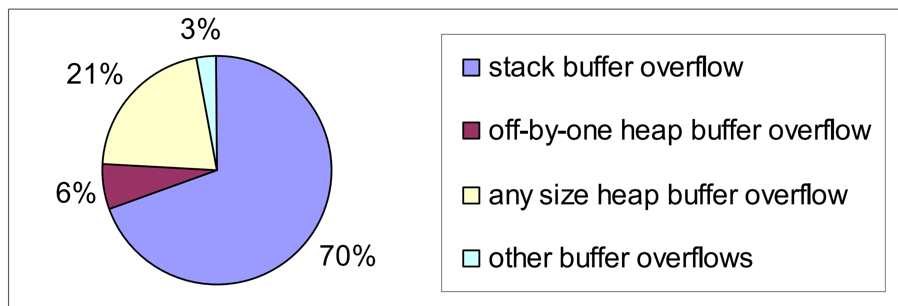
之前的實驗與討論知道當程式中緩衝區使用的數目越多，受 BODAR 保護的應用程式浪費的記憶體資源也會越多。我們嘗試在保護的範圍與記憶體資源間做取捨來改進這個問題。

首先針對緩衝區的特性做分析，我們認為堆疊緩衝區通常於函式開始時配置而在函式結束時釋放，不會長時間佔用記憶體資源，所以針對此類緩衝區做保護，記憶體浪費的總量不會太多。

至於其它的緩衝區，越大的緩衝區相對浪費的空間越少，因此大型緩衝區仍以 BODAR 方式保護，小型緩衝區則依舊有的 malloc 方式：盡可能在分頁中配置許多小型緩衝區。我們以 512 位元組作為大型或小型緩衝區的分界。此外，在配置小型緩衝區時我們將配置的大小增為原來的四倍，原因是有一種特殊類型的緩衝區溢位漏洞稱做 off-by-one 漏洞，這類漏洞溢位的量都不大而且有限，故配置時多出來的三倍空間應足夠容忍 off-by-one 漏洞溢位的資料。雖然這樣喪失了偵測這類緩衝區漏洞的能力，但仍能有效的保護應用程式不會因溢位造成阻斷服務的問題。



(a)



(b)

圖 4-4　緩衝區溢位漏洞的特性分析。(a) 自 2002 年至 2005 年 SecuriTeam 的漏洞特性分析圖。(b) 2005 年 CVE 的漏洞特性分析圖。

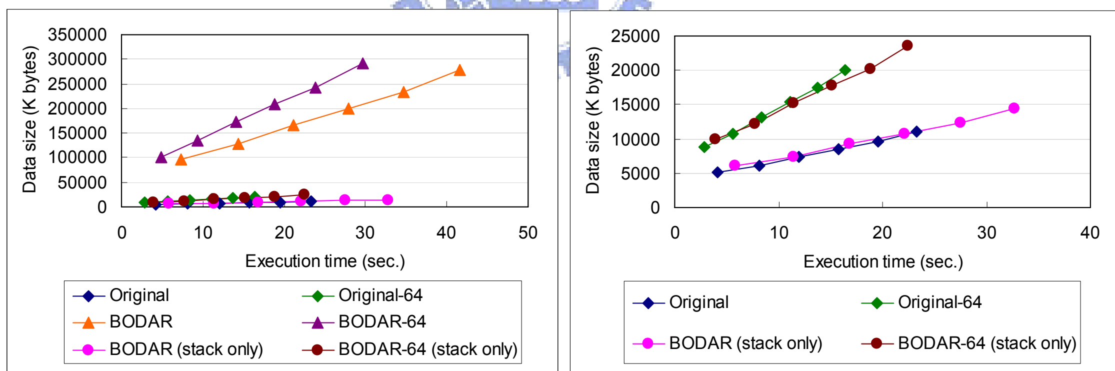


圖 4-5　gawk 以 BODAR 折衷方式保護與其他版本的結果比較。右圖為左圖移除原來 BODAR 版本的結果。

圖 4-4 為我們調查 SecuriTeam 與 CVE 中不同特性的緩衝區溢位漏洞所佔比率。結果顯示 BODAR 仍然保有約 73%與 76%的保護能力。而圖 4-5 為 gawk 分別引入原來的 BODAR 與改進過的 BODAR 執行 S 腳本程式的效能與記憶體資源使用的比較。可以看到改進過的 BODAR 版本執行效能約

為原來 BODAR 版本的 80%，記憶體使用量約為原來 BODAR 版本的 6%，並且明顯地緩和了原先記憶體使用量增加過於快速的情形。

我們亦可將 BODAR 的原來方案與折衷方案整合在一起，以已配置的記憶體總量為門檻，低於此門檻時仍採取原先的 BODAR 方案保護，高於此門檻則以折衷方案保護。以我們實驗中 gawk 最糟糕的組合（執行 S 腳本程式並輸入 30 萬行的紀錄檔）來看，單一 BODAR 保護的記憶體使用量為 277404 KB，若門檻設為 10000 KB，則記憶體使用量降為 23676 KB，是原來的 8.5%左右。圖 4-6 為 gawk 在 BODAR 整合方案保護下執行 S 腳本程式的實驗結果，可看到當檔案逐漸增大，需使用越來越多記憶體時，BODAR 整合版本與原來版本的使用量差距將逐漸縮小。

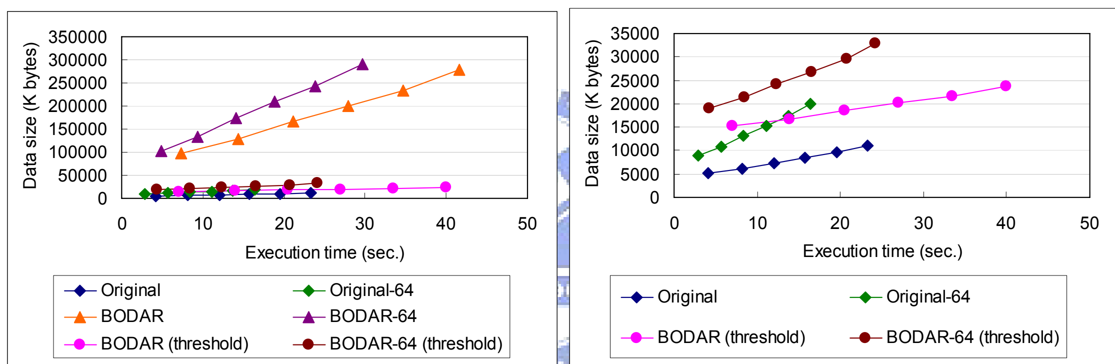


圖 4-6 gawk 以 BODAR 整合方式保護與其他版本的結果比較。右圖為左圖移除原來 BODAR 版本的結果。

第五章 結論與未來發展

BODAR 利用現今作業系統普遍支援的硬體分頁保護機制來達到事件驅動方式偵測。這個分頁保護機制為當存取指令試圖存取未配置的虛擬位址時，會促使作業系統產生 SIGSEGV 訊號。我們將緩衝區之後跟隨一塊未配置區域，使緩衝區溢位會存取到未配置區域而發出訊號，訊號將觸發溢位處理函式進行恢復動作。

BODAR 的恢復概念來自於 BMB，同樣的透過容忍緩衝區溢位資料來達到維持程式的執行。應用程式在 BODAR 保護機制下，每個緩衝區後將跟隨一塊稱為守衛區域的未配置區域，可用來容納緩衝區溢位的資料。當緩衝區溢位發生時，溢位處理函式會解析程式試圖存取的記憶體位址並在其上配置一塊實體記憶體，在處理完後溢位存取將會在新配置的記憶體上執行，不會影響到其他在記憶體中的資料，程式將可繼續執行，直到下一次緩衝區溢位。

BODAR 的作法相比於 BMB 的實作方式，BMB 對溢位資料是是採取另找一塊離散的空間存放，這表示之後每次溢位存取，非得經過位址轉換，與 BODAR 直接將溢位資料置於緩衝區之後，溢位資料可直接連續存取，不需再經任何手續，因而在恢復的速度上 BODAR 會有較佳的表現。

我們透過實驗驗證了擁有恢復機制的重要性。一個無恢復能力的緩衝區溢位保護機制即使引入到服務程式中，當遇到持續不斷緩衝區溢位攻擊時所造成的阻斷服務時，效能將嚴重下降。BODAR 具有的快速恢復能力將使受其保護的程式提升在攻擊下的生存能力與持續提供服務能力。

此外由於 BODAR 會造成記憶體資源快速浪費的問題。我們也提出了一個折衷方案，考量緩衝區溢位漏洞的特性，藉由犧牲少部分的保護範圍換取大量縮減記憶體使用量與效能的提升。

未來將朝不縮減保護的範圍而能降低記憶體使用量繼續做改進。一個可能的方法如圖 5-1。

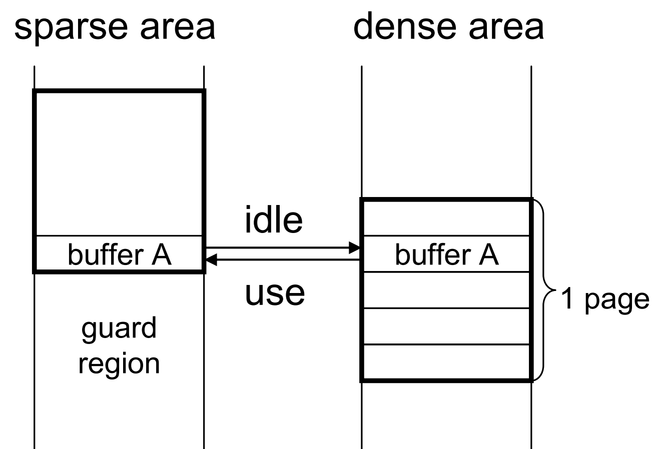


圖 5-1　未來可能的改進方法

整個記憶體將分為離散區段與緊密區段，離散區段仍以 BODAR 的方式配置緩衝區，而緊密區段以 chunks 的方式配置小型緩衝區。在理想的狀態下，一開始小型緩衝區將配置在離散區段，當系統發現這個小型緩衝區沒使用時，會將小型緩衝區備份至緊密區段並將小型緩衝區所佔用的分頁釋放，當要存取小型緩衝區資料時，再於原來的位置重新配置分頁並將備份的資料回存，透過這樣的操作方式達到存放時不佔用記憶體，使用時能有 BODAR 保護的目的。

參考文獻

[1] ACME Laboratories. thttpd. http://www.acme.com/software/thttpd/ (last access: May 2005).

[2] Aleph One. Smashing the stack for fun and profit. http://www.insecure.org/stf/smashstack.txt

[3] A. Baratloo, N. Singh and T. Tsai. "Transparent Run-Time Defense against Stack-Smashing Attacks." In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pp. 251, June 2000.

[4] Beyond Security. SecuriTeam. http://www.securiteam.com/ (last access: May 2006).

[5] S. Bhatkar, D.C. DuVarney and R. Sekar. "Address Obfuscation: An Efficient Approach to Combat A Broad Range of Memory Error Exploits." In *USENIX Security Symposium*, pp. 105-120, August 2003.

[6] C. Cowan, S. Beattie, J. Johansen and P. Wagle. "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities." In *In Proceedings of the 12th USENIX Security Symposium*, pp. 91-104, August 2003.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang. "StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks." In *Proceedings of the 7th USENIX Security Symposium*, pp. 63-78, January 1998.

[8] C. Dahn and S. Mancoridis. "Using Program Transformation to Secure C Programs Against Buffer Overflows." In *IEEE Proceedings of the 2003 Working Conference in Reverse Engineering (WCRE'03)*, British Columbia, Canada, pp. 323-332, November 2003.

[9] D. Dhurjati, S. Kowshik, V. Adve and C. Lattner. "Memory safety without runtime checks or garbage collection." In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, June 2003.

[10] N. Dor, M. Rodeh and M. Sagiv. "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C." In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[11] J. Foster, M. Fahndrich and A. Aiken. "A theory of type qualifiers." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[12] Free Software Foundation Inc. GCC. http://gcc.gnu.org/ (last access: May 2006).

[13] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference A-M. http://developer.intel.com/design/pentium4/manuals/index_new.htm (last access: January 2006).

[14] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney and Y. Wang. "Cyclone: Asafe dialect of C." In *Proceedings of the USENIX Annual Technical Conference*, pp. 275-288, June 2002.

[15] R. Jones and P. Kelly. "Backwards-compatible bounds checking for arrays and pointers in C programs." In *Third International Workshop on Automated Debugging*, May 1997.

[16] S.C. Kendall. "Bcc: run-time checking for c programs." In *Proceedings of the USENIX Summer Conference*, 1983.

[17] L. Lam and T. Chiueh. "Checking Array Bound Violation Using Segmentation Hardware." In *In Proceedings of the 2005 International Conference on Dependable Systems and Networks(DSN'05)*, May 2005.

[18] D. Larochelle and D. Evans. "Statically Detecting Likely Buffer Overflow Vulnerabilities." In *Proceedings of the 10th USENIX Security Symposium*, pp. 177-190, August 2001.

[19] Mindcraft Inc. Webstone. http://www.mindcraft.com/benchmarks/webstone/ (last access: January 2006).

[20] G.C. Necula, S. McPeak and W. Weimer. "CCured: Type-Safe Retrofitting of Legacy Code." In *Proceedings of the Principles of Programming Languages (PoPL)*, pp. 128-139, January 2002.

[21] Pax. http://pageexec.virtualave.net .

[22] B. Perens. "Electric Fence Malloc Debugger." In *Pixar Animation Studios*, 1993.

[23] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy and T. Leu. "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other

Memory Errors)." In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004.

[24] R. Rugina and M. Rinard. "Symbolic bounds analysis of pointers, array indices, andaccessed memory regions." In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pp. 182-195, June 2000.

[25] O. Ruwase and M.S. Lam. "A Practical Dynamic Buffer Overflow Detector." In *Proceedings of the 11th Annual Network & Distributed System Security Symposium*, pp. 159-169, February 2004.

[26] SecurityFocus. Mod_mylo apache module REQSTR buffer overflow vulnerability. http://www.securityfocus.com/bid/8287/ (last access: January 2006).

[27] SecurityFocus. Thttpd defang remote buffer overflow vulnerability. http://www.securityfocus.com/bid/8906/ (last access: January 2006).

[28] S. Sidiroglou, G. Giovanidis and A. Keromytis. "Using Execution Transactions to Recover from Buffer Overflow Attacks." Technical Report Technical Report CUCS-031-04, Columbia University Computer Science Department, September 2004.

[29] Solar Designer. Non-Executable User Stack. http://www.openwall.com/linux/ (last access: May 2006).

[30] The Apache Software Foundation. The Apache HTTP Server Project. http://httpd.apache.org/ .

[31] The MITRE Corporation. Common Vulnerabilities and Exposures. http://cve.mitre.org/ (last access: May 2006).

[32] The Software Technology Laboratory, Queen's University, Kingston, Canada. TXL Home Page. http://www.txl.ca/ (last access: January 2006).

[33] Vendicator. Stack shield. http://www.angelfire.com/sk/stackshield/ (last access: January 2006).

[34] D. Wagner, J.S. Foster, E.A. Brewer and A. Aiken. "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities." In *Network and Distributed System Security Symposium*, pp. 3-17, February 2000.