國立交通大學

資訊科學與工程研究所

碩 士 論 文

以版本歷程與動態執行資訊產生註解文件之自動化系統

Automatic Program Document Extraction

from Revision History and Run-time Trace

研 究 生：黃建智

指導教授：黃世昆　教授

中 華 民 國　九 十 五　年 七 月

以版本歷程與動態執行資訊產生註解文件之自動化系統

Automatic Program Document Extraction
from Revision History and Run-time Trace

研 究 生：黃建智　　Student：Chien-Chih Huang

指導教授：黃世昆　　Advisor：Shih-Kun Huang

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

以版本歷程與動態執行資訊產生註解文件之自動化系統

學生：黃建智　　指導教授：黃世昆

國立交通大學資訊科學與工程研究所　碩士班

摘要

文件撰寫工作在專案中是必要而不可或缺的過程，但我們卻常忽略其在發展過程中的必要性。對於一個組員時常變動的團隊，在沒有明確註解文件的情形下難以將程式的修改部份加入其系統。本研究期望透過動態分析的技巧以及版本資料庫的內容此二途徑，以揭露專案開發的內隱知識。我們的目標是在不需程式內特殊標註的情形下，自動產生文件(documentation)以增進人員對程式的理解，或者是以此自動產生之文件對原始碼除錯。此文件以程式常用模式(frequent pattern)表示，方法上使用 sequential pattern mining 演算法於動態歷程(run-time trace)上找出 pattern，以出現次數(support)、機率(confidence)描述該 pattern，另外使用版本資料庫之資訊，得出程式中每個識別符(identifier)的變動情況(variant)。使用上述三種參數決定 pattern 的品質，結果可分爲三類－Frequent、Potential-Error、Unlikly。開發者可依此文件得到程式撰寫建議，或是提示原始碼中可能出錯的部份，並指出可能的正確撰寫方法。

關鍵字：　程式文件, 版本歷程, 動態分析

Automatic Program Document Extraction

from Revision History and Run-time Trace

student: Chien-Chih Huang Advisor: Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Documentation is an essential and indispensable part in a software project but we often ignore its importance for the development process. Developers construct the software systems with incomplete comments or features left undocumented; for this reason, it's hard for a rapid changing team to accommodate changes inherently embedded without explicit annotations.

Via the techniques of dynamic analysis and revision history manipulation, we discover then implicit knowledge in the projects development. Our main objective is to generate program documents without explicit specifying related annotations in the program source. The knowledge of program understanding could be also used to debug the source code.

The document is represented as program frequent patterns. We apply the sequential pattern algorithm on run-time trace to find out the patterns and describe patterns by occurrence times(support) and probability(confidence). We also process the revision history database to count the change times for each identifier and this value are named after "variant".

Patterns are ranked by the above 3 parameters. We classify the pattern into 3 categories: Frequent, Potential-Error and Unlikly。Upon this document, developers could obtain programming suggestion. In addition, our system could point out the susceptible site in the source and prompt the possibly correct coding style.

Keywords: Program Document, Revision History, Dynamic Analysis.

誌謝

撰寫這份論文的過程雖然艱辛，但最終還是完成了，若非黃世昆老師的督促與建議，在低潮時給予策進與鼓勵，否則這份論文恐怕是無緣面世，由衷地感謝老師在協助論文撰寫以及最後的呈現。

撰寫這份論文的過程使我對於程式安全這個領域有了基礎的了解，這要感謝老師以及眾同學們辛苦建立的實驗室，有了你們的討論與建議，才能使我對這個領域有更全面的了解，也感謝大家每週一次又一次地聽著我的研究並給予建議。

頭一次離家在外求學要感謝許多人的幫忙，特別是黃老師除了在研究上外，在生活上也不吝於給予關心與指導，而實驗室眾同學們給予我的幫忙更是難以計數，在修課時也認識了不少的朋友，大家的幫助讓苦悶的研究生生活變的充實。

最重要的是感謝父母的支持，讓我得以無後顧之憂地在學業上努力，在我低潮時能夠給予鼓勵，謝謝你們。

目錄

表目錄

圖目錄

1. 緒論

1.1. 研究動機

軟體發展過程中，由於團隊成員的更迭，使得舊成員無法再維護其先前所負責的部份，後繼者僅能依循之前所留下之文件及程式碼。在大部份的情形下，程式設計者主要的工作是使程式動作能符合需求，而非文件撰寫，在時間與資源的壓力下，文件撰寫常成爲犧牲的部份，因此在這種情況下，文件的品質是無法確定的；而在程式的了解上，我們僅能由檢視程式碼或再加上 Cross-Reference[1]工具的協助，然而在一個大型專案中，程式元件之間的互動可能是相當複雜的，使用這些靜態檢視或分析的工具僅能有部份協助，在程式語意的部份需要更進一步的工具來告知。 我們將爲上述問題提出解決方法：當文件缺乏而程式碼複雜的情形下，新進者如何更快地參與程式的開發。

經由這套方法的幫助，可進一步地解釋部份軟體工程中的現象。這項工作有益於以下的情況：在開放原始碼專案中，有時需取出特定檔案、與其發展歷程中的狀態，做一分支開發或其它工作；或整合前面的發展時，特別是之前的程式碼並非由自己所撰寫，必須判斷之前的修改意義，才能進一步地整合成完整的程式。

當程式整合之後，開發者需判斷程式的非預期行爲，但每個專案都有其特有之問題，這些問題的發現或解決有賴於該專案團隊的經驗。在舊成員離開後，這種經驗可能無法傳承。透過我們的方法，期望將這些經驗以某種自動化的程度呈現出來。

當程式設計人員因爲種種因素－如許久未參與或更動該部份之開發，亦或是新加入者－無法獲得該經驗時，首先可能是程式開發產生阻礙，甚至是開發者在一知半解的情形之下撰寫或修改該部份，可能會導致違反系統內隱含的規則(implicit rule)，此類規則爲程式正常執行的條件，因此在某些情形下違反該規則之設計會導致程式出錯，嚴重者可能會導致系統毀損。因此，在此我們前段所得之結果爲基礎，針對程式中可能出錯的部份提出建議，供程式撰寫者參考。

1.2. 問題描述

在 Imperative language paradigm（如 C, Pascal 等語言）之程式開發中，二個或二個以上的函式(function call)有時會共同出現於程式中，此種情形在出現次數達到一閥值後，便可將此共同出現之情形稱爲 pattern。目前現有的論文大多注重於單純函式呼叫的共同出現性[2-5]，而較少關注到函式所使用之參數或回傳值。

在此我們發現到，函式使用常伴隨著暫態的使用，此種暫態有時以區域或全域變數的形式出現。在此以一段 C 語言程式說明：

{ ...
var_DEP = foo(var_ARG);
buk(var_DEP);
... }

其中 var_DEP 以及 var_ARG 皆爲上文所稱之暫態，而 foo 及 buk 各爲函式呼叫。

函式 foo()所回傳的值—var_DEP 爲函式 buk()所用，在出現頻率夠高的情形下，我們可以將此種使用情形稱爲 pattern。在本篇論文中，我們提出演算法以找出此類 Pattern，用以補[2-5]在 dataflow 分析上之不足。

1.2.1. 以 OPENSSH 爲例

我們取 OpenSSH （版本爲 4.3p2），位於 ssh.c 內的 control_client()函式中一段程式做爲說明：

```
1282        if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
1283                fatal("%s socket(): %s", __func__, strerror(errno));
1284
1285        if (connect(sock, (struct sockaddr*)&addr, addr_len) == -1) {
1286                if (mux_command != SSHMUX_COMMAND_OPEN) {
1287                        fatal("Control socket connect(%.100s): %s", path,
1288                            strerror(errno));
1289                }
1290                if (errno == ENOENT)
1291                        debug("Control socket \"%.100s\" does not exist", path);
1292                else {
1293                        error("Control socket connect(%.100s): %s", path,
1294                            strerror(errno));
1295                }
1296                close(sock);
1297                return;
1298        }
```

圖 1 OpenSSH 程式內容舉例

以上圖爲例，此程式使用於 openssh 之 client 中，功用爲發起連線。當中可以看到第 1282 行之 socket()，第 1285 行之 connect()以及第 1296 行之 close()此三個函式呼叫，使用傳統使用資料探勘的程式分析方法，只能注意到此種 Pattern－僅包含函式呼叫，但是[2]之研究注意到，變數也可成爲 program rule 之中的重要成分，在上例中即爲 sock 此變數的使用，但[2]的方法是將函式及變數視爲相同層級之分析實體(program entity)，但並沒有注意到其中的依賴(dependency)關係。

在此例中可以看到：

在 1282 行中函式 socket()將其回傳值置於 sock 此變數中，而

在 1285 行中函式 connect()以參數的方式讀進變數 sock。

在 1296 行中函式 close()以參數的方式讀進變數 sock。

以下即爲所描述之圖示，這也是本篇論本所希望找出的 pattern。

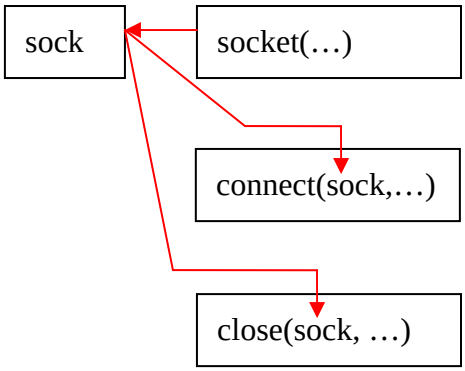


圖 2 Dependency 示意圖

因此變數 sock 依於 socket()，而 connect()及 close()依於變數 sock，因此在此可以推斷 connect()及 close()的動作可能會依著函式 socket()每次回傳值的變異而有所不同。因此，將變數 sock 以及函式 socket()，connect()以及 close()視爲同一 pattern 爲一合理的歸類法。

1.2.2. 使用專案版本庫

軟體專案的版本庫記載著專案原始碼中各種變動訊息，其中各個版本間的變動差異可能是該專案的特殊情形，舉例來說，[6]分析各專案的版本資料庫以得到三種規則：

一．當程式中有部份的程式碼被修改時，自動預測或建議其它應共同修改部份；

二．在新版本加入後，依據以往的更新紀錄，偵測此次是否爲不完整的更動；

三．傳統上對單一版本的程式分析有其弱點，特別是針對程式中不同 program entity 之歸類（coupling），加上版本資料庫的分析可以補其不足。

在本論文中，我們希望利用版本變動的部份特性以揀選出值得注意的 pattern，這些 pattern 中的單元可能是在以往在版本變動時曾發生過缺失，若能將此 pattern 予以挑出，則在錯誤警示的功能上可以有更高的可信度。

1.3. 研究前提

從參考文獻與研究者的觀察中，這份論文設定下列前提，以此前提發展後續之研究：

1. 有連帶呼叫性質的函式有時會使用相同的變數，如圖 2 Dependency 示意圖圖 2 所揭示之的變數 sock，故若能以此變數爲出發，則應可找出使用此變數同一組的函示。
2. [7]的研究指出幾個現象，首先是５０％以上的版本庫更動行數在十行以下，而其中在只有一行更動的行爲中，只有４％會造成以後程式的錯誤，且只有２．５％是在完整地增加新功能，由此二原因我們推論，若大部份的一行更動不是發展新功能，那麼它便可能是在修改舊程式，而修改舊程式的行爲可合理地推定爲修正程式的錯誤。
3. 本研究將 pattern 限定在同一個函式呼叫的範圍內，即 intra-procedural。若採用

inter-procedural 之分析，則一來技術上會有相當大的困難，二來使用者反而會因過多的資訊造成使用上的困難，使用 intra-procedural 分析則可以簡化其理解上的複雜度。

1.4. 研究目的

綜合以上而言，這份論文希望達到幾個目的：

一是使用動態分析與 dataflow 分析，融合資料探勘的技術，針對特定專案找尋其常用之 pattern，而 pattern 要能有以下的二個要求。

a. 找出的 pattern 可以明確的表達出當中元素執行上的先後次序；
b. 能夠反應出 pattern 內不同元素(program entity)之相依情況。

二是以版本資料庫爲基礎，將高重要性的 pattern 挑出，供程式開發者建議。

1.5. 章節架構

本論文之第二章爲近年來相關之研究，並以不同的研究途徑加以分類、比較，著重在各論文使用之方法、優缺點及結果比較，後半段爲本研究所使用之相關技術之說明。第三章描述本論文所使用之方法，分爲三個階段－取得分析資料、建立常用樣式以及常見錯誤建議。第四章展示將此論文之方法應用於 OpenSSH[8]上。第五章總結本文之貢獻與未來可能之研究方向。

2. 相關研究

以下我們依使用技術的不同，分爲幾個大類介紹和本篇論文相關的研究。其中第一節至第三節爲文件產生(documentation)相關之技術，而第四節及第五節爲本論文所使用之相關工具，第六節爲總結第二章之討論，在比較過上列各項相關研究後，決定本論文所採用之方法與工具。

2.1. 直接於程式碼中抽取相關資訊

該類方法是直接分析程式碼，分析技巧比較單純。分爲二類：

一是抽取程式碼中特定格式的註解[9]，此類工具常見的有 Javadoc 或 Perldoc 等，此法最爲簡單，然而缺點在於：

- 文件的品質取決於註解撰寫者，每個人的撰寫風格與對程式的了解不一，因此無法確保註解的品質。
- 若不注意於維護註解，當程式變動時則可能發生註解與實際程式內容不一致的情形。
- 註解通常只能針對單個 API 說明，缺乏整體概觀的說明。

二是使用 Cross reference 工具[1]，其將程式碼中每個函式、變數明確地告知其定義處或使用處，使用者可快速得知程式中的細節；然而缺點也因其呈現方式太過細微而無法得知程式的整體概觀。實際使用情況如下圖，在資料庫已建立完備的情況下，當使用者輸入函數名或變數（圖中輸入爲 BUG_ON），該工具即可顯示出其定義處，以及在相關檔案中使用之情形，在本例中可看到 BUG_ON 在四個地方被定義，而在 526 個檔案中被使用，以及其確切行數。

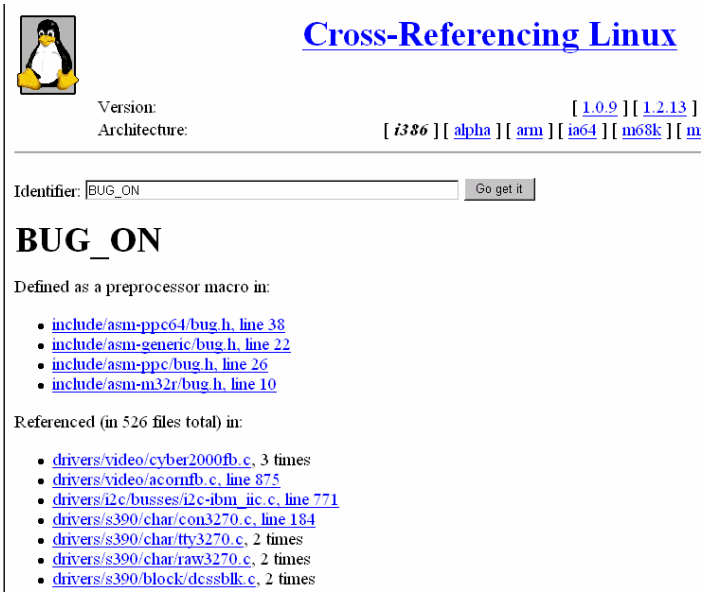


圖 3 Corss-Reference 工具使用圖

2.2. 版本資料庫

以下分爲兩個部份討論，首先將簡單介紹 SCM（Software Configuration Management）之概念，其後再介紹結合 SCM 之程式分析之研究。

2.2.1. SOFTWARE CONFIGURATION MANAGEMENT

在軟體專案發展中，程式碼會不停的地修改，此時便需要一架構用以協調多人、多時之發展過程，SCM 即用以處理此問題，Zeller[10]的研究中定義 SCM 爲一套方法用以組織及控制演變之系統，其具備以下七類功能：

- Identification－辨認出系統內之各部份(component)，使其能被明確存取；
- Control－控制專案的版本並確保其與初版(basline)對應關係之一致；
- Status Accounting－紀錄各元件之變動過程並統計；
- Audit and Review－確保專案之完整性，即各元件間不相衝突；
- Manufacture－以較佳的方式產出專案；
- Process management－確保專案生命週期(project life cyclte)與開發團體之政策(policy)一致；
- Team work－協調多人發展

在版本變動紀錄的功能上，直觀的方法乃是 Checkin/Checkout Model，其完整紀錄各版本之內容，如 git 即採用此種方法，此方法雖然簡單且存取快速，然而會造成需要較大的儲存空間。另一種目前常見的方式乃是 Change-Oriented Model，在其系統內僅完整留存最新或最舊之一版，視不同的實際系統而定，其它的版本則以差異(diff)的方式存在，在此我們簡單地介紹 diff 之概念。

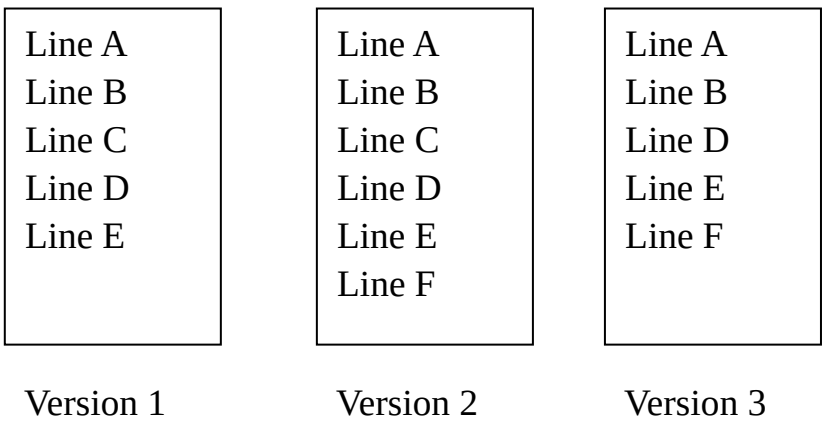


圖 4 diff 示意圖

如上圖所示，version 1 與 version 2 之差異僅在 Line F，但若僅紀錄“*增加 Line F*”則無法由 version 1 建立正確之 version 2，因爲 Line F 可能在 Line E 之後，也可能加在 Line A 之後等其它地方，故尚需記錄定位點；CVS 的做法是以前一版的行數爲定位點，

紀錄下變動的行號與行數，因此若比較 version 1 與 version 2 則會產出

add 5 1

Line F

此表示在 version 1 之第五行之後加入一行 *line F*，同樣地，若比較 version 2 及 version 3 則以下列方式表示：

del 2 1

此表示在 version 1 之第二行之後刪去一行，由於刪去的內容在新版中已不復存在，因此不需在 diff 中紀錄被刪除之內容，若要取得被刪除之內容，只需在舊版中找到相對應的行數（此例爲第二行）即可知道變動內容。

由上例可顯示，若只紀錄差異，則將會比紀錄完整版本更能節省儲存空間，但缺點是當取出版本距離基準版越遠時，因爲要不斷地計算各版本間之差異，系統之效率會越差。

目前常用的 SCM 系統有 CVS (Concurrent Version System)或 SVN (subversion)[11]，SVN 爲 CVS 之改進，此二者皆爲 change-oriented model。由於歷史緣故 CVS 仍較爲普遍，目前大部份著名的開放原始碼專案仍使用 CVS，而本研究所分析之 OpenSSH 專案仍是以 CVS 之形式存在，應該第三章的方法設計將以 CVS 做爲主要的處理目標。

2.2.2. 應用版本資料庫之文件製作

目前已有多種研究利用版本資料庫(revision history database)做爲分析工具，黃與劉[12]在其研究中透過對檔案與目錄層級的分析，判別出重要的目模組（即目錄），依此進而界定開發者在發展過程之重要性，但是該研究並沒有處理函式層級以下之分析；Cubranic[13]則是將開發過程中的各項資訊予以連結，例如將錯誤回報庫(bugzilla)與版本變動差異(diff)建立關係，當使用者查詢如函式或變數名稱時，該系統可依相關度列出曾發生過的相關歷史事件，雖然已針對個別檔案內部進行部份分析，但並沒有針對各程式語言的特性發展，反而像是關鍵字搜尋的方式，在使用時仍可能陷於大量的資料中。

另一類的研究結合版本資料庫與程式分析，[14]揭示在進行此工作時應有的預備工作以及注意事項，以該研究爲基礎，Zimmermann 等人使用 association rule（於本章末節介紹）之分析方法找出程式碼中常被同時修改的 program entity，此 program entity 常是函式或變數，然而該文並無針對函式及變數的不同特性設計方法，且使用 association rule 的方法無法描述 entity 在程式內之先後關係（於本章末節將會詳細說明）。另外[2, 3, 5]則同樣使用 association rule 之方法，其中[3, 5]用以找出程式中具相關的函式組，而[2]則將變數融入函式相關性的分析中，詳細的內見下表比較。

最後介紹的方法則不使用 association rule 分析，[4]直接比較不同版本之 AST 以了解程式內部細項(entity)之演變情形。

表 1 結合版本庫之方法比較

	標的	Program entity	使用工具	分析單位 (granularity)	結果示例	方法描述
Guide Software Changes[6]	各種電腦語言		ROSE (Reengineering Of Softwave Evolution)	Each revision diff	{processor_cost} ➔ {i386_cost, i486_cost, k6_cost} 當 processor_cost 修改時，則 i386_cost、i486_cost 及 k6_cost 也會跟著修改	They process the revision history database to fetch the fetch each revision and then group changes in various files into one transaction. Rules are mined from these transactions. The rules and their supports and confidences (parameter in data mining algorithm) of the rules are provided to the users together.
System specific Rule[5]	C 語言	函式	Edison Design Group C parser	10 lines in a revision	{GlobalLock}➔{GlobalUnlock} 當 GlobalLock 加入時，則預期 GlobalUnlock 會在十行內加入。 此論文限制其 pattern 長度爲 2	Mining pair-wise pattern for two types: *called after* and *conditionally called after.* The former is simple pair-wise call pattern. But the later introduced the branch concept. For example, {…var = func_a(…); if (var==0) then func_b(…);} func_b is triggered by var, which is the return value of func_a. Repository is used for testing whether each pair is steadily added in the same revision.
Matching Method calls[3]	JAVA	函式及變數	文中未提實際工具，但仍使用 AST 做爲分析	each revision diff	同上 除 pattern 次數外，加入機率之概念以判	By applying associative rules, they find pair-wise patterns in Static phase. For corrective ranking, furthermore, either function in a

			的中介。		定 pattern 之可用性	pattern must be one-line check-in at least once. ***Execute the program*** and analyze the ***dynamic*** traces to see whether the patterns are violated or not. The Corrective ranking is more effective than regular ranking.
PR-Miner[2]	C 語言	函式	(modified) GCC	intra-procedure	{scsi_host_alloc}➔{scsi_add_host, scsi_scan_host}當 scsi_host_alloc 加入時，則預期 scsi_add_host, scsi_scan_host 將會同時加入	Some implicit programming rules can be represented as the set of execution pattern of program elements. In this paper, the pattern size is not limited to 2 program element but also they mined both function and variable as program element.
AST Matching[4]	C 語言	AST 中各節點	CIL	each revision	統計 program elements（包含 structure, function call 及 variable）之修改、刪除、增加或更名。	To verify whether the program elements are added, changed, deleted or renamed in successive revision. These elements are structure, function, variable, typedef. They diff ASTs in successive revisions.
本論文	C 語言	函式及變數	Cvsup 及 CIL	Each revision diff	後述	後述

2.3. 逆向工程

逆向工程(reverse engineering)爲一廣泛使用之概念，Sommervile[15]定義其爲*分析程式以取得資訊，並用以紀錄(document)程式之架構或功能*。目前已有相當多之研究，其中我們列出幾篇與本文相關研究

在 Whaley[16]的研究裡提出一套架構用以萃取並建立程式 api（application program interface）呼叫關係之有限狀態模式（FSM, Finite State Machine），其中各 api 爲 FSM 之節點(node)而呼叫關係則爲有向邊（directed-edge），其方法是以動態程式分析之方法先紀錄程式行爲，再加以分析得出 FSM，最後以靜態分析方法檢視原始碼中不符合此 FSM 之行爲。與我們的研究相比，我們的重點不在於提出完整的程式構件(component)間的呼叫關係，而在於驗證 FSM 圖中每一 path 之正確性。

另外[17, 18]則是將程式還原至 design pattern，其中[17]以開發嵌入式系統爲例，說明在描述 design pattern 時可分爲那些類別，以及該注意的記錄事項。而[18]則是使用靜態分析的方法建立 pattern，並以 DPML(Design Pattern Markup Language)表示，再與原程式之 ASG（Abstract Semantic Graph，由靜態分析而來）比對，確認 pattern 與 source 之關係。此類的研究以簡單的程式分析方法，提出並應用其描述系統，如何能夠完整地描述 pattern 爲其重點，在 pattern 找尋上並沒有太多獨特的技術，但其在表示 pattern 的方法上可應用於本研究上。

Ernst[19]之研究則使用動態分析方法建立 program specification，再以靜態方法驗證，此研究特別的是透過對程式執行中的不變數(invariant)分析而做爲定位點，進而分析函式以得到 program specification，與大多數的研究只使用函式做爲分析主軸不同。雖然我們的研究不使用不變數之分析，但是導入變數的 pattern 分析則爲我們的研究提供一新的發展途徑，在第三章中將會描述如何將變數融合在以函式爲主軸的程式分析。

另一類的研究則是將程式還原成 UML(Unified Modeling Language)，UML 是一套描述辺法，使用圖型化的方式描述程式或系統的抽像結構，內含多種類的圖表。此類研究極爲眾多，而市面上也已有商品化的工具(UML Case Tool)可取得，此類工具大致以靜態分析之方法爲主。[20]介紹並比較幾種可產生 UML 之 CASE tool－如 Together、ROSE、Fujaba 及 Idea；[21]則使用動態分析得到 UML 內之 sequence diagram，其獨特處是使用 OCL(Object Constrains Language)以描述執行歷程與 sequence diagram 之間的關係。我們的研究不在於產生 UML，然而，透過適當的轉換，我們的研究結果可用於產生 UML 上。

最後一類的研究是分析程式執行時之資料結構(Data structure)做爲 Documentation，其中[22]用以協助用者檢視記憶體中程式所建立之資料結構，並以圖型化的方法表示記憶體區塊之大小或彼此關係，可用於了解程式結構或除錯；而[23]則是以自動化地方式，以程式分析及部份人工輸入所建立之模式，針對資料結構除錯。

在此將上述各方法比較如下表：

表 2 逆向工程相關研究比較

	靜態分析用途	動態分析用途	標的	使用工具	結果呈現方法
Automatic extraction of object-oriented component interfaces[16]	使用 model 除錯	建立 model	JAVA	Byte Code Engineering Library	Finite State Machine
Mining design patterns from C++ source code[18]	建立 Abstract Semantic Graph 及 DPML	無	C++	Columbus (for ASG)	Design Pattern Markup Language
Automatic generation of program specifications[19]	驗證 model	建立 model	JAVA	DAIKON (for invariant analysis) ESC (static analysis)	Annotation in Source
Towards the Reverse Engineering of UML Sequence Diagrams[21]	instrumentation	建立 model	C++	PERL (for instrumentation) JAVA (convert trace to sequence diagram)	Sequence Diagram (UML)
DDD—a free graphical front-end for UNIX debuggers[22]	Instrumentation	尋找記憶體單元間之關係	C	GDB (for backend) Gtk+ (for frontend)	Data Structure Graph
Data structure repair using goal-directed reasoning[23]	建立 model	驗證實際執行是否符合 model	C	JAVA, C	Bug highlight
本論文	Instrumentation	紀錄 trace	C	CIL, SPAM, python	Program rules

2.4. 程式分析相關

2.4.1. 程式分析工具

本論文使用的方法（於第三章詳述）需依賴程式分析工具的使用，但發展此類程式並非此研究之重點，且目前已有相當多的分析架構可供使用，因此選擇使用或修改現有的工具。我們選擇前幾節已使用的或常見的工具加以分析比較：

- GCC

GCC 爲一 open-source compiler，目前普遍使用的程式語言，如 C、C++、JAVA 等皆有支援，有一些研究以修改 GCC 的方式進行，而事實上 GCC 已提供了相當多的輔助功能，以本研究相關的的功能來說，約數十行的程式撰寫量即能紀錄函式在執行時之呼叫情形，然而本研究尚需變數之分析，而 GCC 並不提供，但由於 GCC 相當龐大，修改 GCC 將可預期到耗時過多，因此本研究不以 GCC 做爲分析之工具，而只單純使用其編譯(compiling)功能於 openssh 上。

- TXL[24]

由 Cordy 所發展之 TXL 主要用於程式分析與跨語言之轉換(language transformation)上，可應用於各種程式語言上，其特點在於可針對程式 AST（Abstract Syntax Tree）中特定之結構進行分析或轉換，在轉換目標明確的情況下，撰寫少量的 TXL 程式碼即可完成工作，然而若需大量精細分析（fine-grained analysis)時，則 TXL 需針對各種不同之 AST 架構撰寫程式碼，其行數隨著分析所牽涉之 AST 複雜度呈指數(exponential)成長，由於本研究需處理及分析變數(variable)，在 AST 中有相當多處可能會出現變數，若使用 TXL 則會需考慮各種變數在 AST 中情況，TXL 程式碼撰寫量將無法負擔，因此本研究在嘗試使用後決定不使用 TXL。

- CIL[25]

CIL 全名爲 C intermediate Language，主要用於分析 C 語言，該系統主要以 ML 撰寫，其主要特點是運用 visitor 於 traverse AST 上，使用者可設定當 visitor 游走至 AST 某類 node 時對程式進行分析或修改，由於 ML 語言規範以及 CIL 本身之設定，其程式撰寫較 TXL 複雜，但由於其能處理 AST 中特定類別的 node，而不需考慮該 node 父節點之形式，此點與 TXL 不同，故不會有需考慮的情況呈指數成長的情況，對本研究來說可符合需求，且不造成程式撰寫耗時過多之情形，因此，本研究最後以 CIL 做爲程式分析之工具。

2.4.2. 分析範圍

如同第一章研究前提中所說明，本研究的分析內容設定於 intra-procedural 而非 inter-procedural，這裡我們以下圖說明此二者之分別：

PROCEDURE H {
CALL A;
CALL E; }
PROCEDURE A {
CALL B;
CALL C; }
PROCEDURE E {
CALL A;
CALL C;
CALL D }

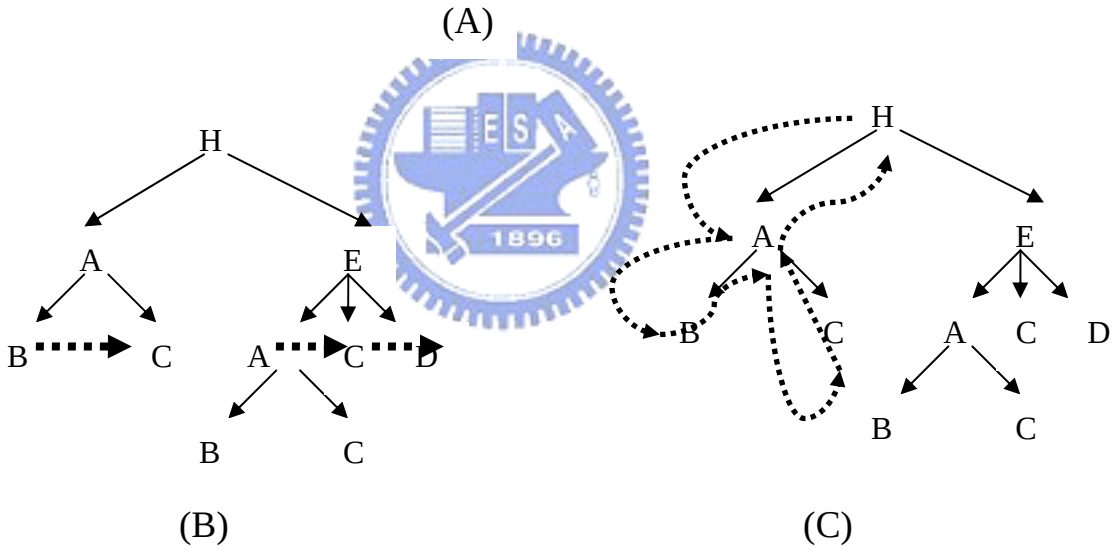


圖 5 程式分析範圍

上圖（A）爲一虛擬碼，表示原始碼中函式呼叫之情形，而圖(B)或圖(C)表示程式實際執行之情況，該樹中各節點（node）表示一函式，而實線之邊（edge）表示呼叫之情形。

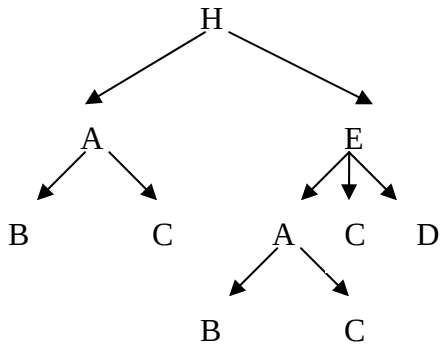
圖（B）爲 intra-procedural 分析方法之示意，虛線表示爲同一個 procedural 內的呼叫先後次序，在此例中可看出於 A 函式中先後呼叫 B 函式及 C 函式，而 E 函式內先後呼叫 A、C、D 函式，但是在 E 函式的呼叫中，我們並不去分析首先執行的 A 函式內部的執行狀況。相對地，經由 inter-procedural 之分析方法可以了解程式中函式的執行序列以及函式呼叫的深度，由圖(C)可以看出 H 函式內會呼叫 A、E 函式，而其中 A 函式又呼叫 B、C 函式。

2.4.3. 分析格式

在決定 inter-procedure 或 intra-procedure 及動態或靜態方法後，便可決定分析之格式，常見的分析格式有以下幾種

PROCEDURE H {
CALL A;
CALL E; }
PROCEDURE A {
CALL B;
CALL C; }
PROCEDURE E {
CALL A;
CALL C;
CALL D }

(A)



(B)

H→A→B→C→E→A→B→C→C→D

(C)

圖 6 分析格式示例

說明： 圖(A)爲一段未經編譯之虛擬程式碼，爲求說明簡化，僅保留函式呼叫之敍述。圖(B)爲該程式碼之實際執行情形。圖中之各節點(node)皆爲函式(procedure)，每個邊(edge)皆爲函式呼叫 (procedure call)，其呼叫順序爲由左至右，由上至下，使用中序表示法（infix）於此樹，即可得時序上之執行序列－如圖(C)

靜態分析分法常使用圖(B)之呼叫樹，分析的範圍若爲 intra-procedure 則爲該樹中各子樹的不同層，而使用動能分析則常用圖(C)之方法，分析的範圍則可使用 intra-procedure 之方法，則圖(C)之 sequence 可分段成符合各 procedure 內容，此例中爲{H}、{A→E}、{B→C}、{A→C→D}、{B→C}五類；或是使用固定長度的 sliding window－如 window size 爲 6 時，其分析的項目爲{H→A→B→C→E→A}、{A→B→C→E→A→B}、{B→C→E→A→B→C}、{C→E→A→B→C→C}、{E→A→B→C→C→D}此五筆。

2.5. 資料探勘技術

本研究的分析對象爲真實世界的專案，其程式量可預期地將會十分龐大，而傳統的統計計數也將因此無法清楚描繪此工作，故在此使用資料探勘(data mining)的技術於程式分析上。[26]認爲資料探勘指的是：

"…extracting or mining knowledge from large amounts of data…"

我們若將程式本體視爲大量資料(large amounts of data)，那麼 extracting knowledge 則是研究動機中稱：尋找開發過程中未揭露之資訊。因此在適當的演算法設計下，使用資料探勘技術自動化地產生專案文件爲一可行之方法。資料探勘內含多種方法，視各應用領域而決定使用一種或多種方法，其中 association rule 相關之方法是用以分析資料內各元件之相關性，本文將採用此方法進行研究。

2.5.1. ASSOCIATION ANALYSIS

而本研究目的在於探討程式中各元件之間的相互關係，因此若使用 Association Analysis 應爲可行，而目前已有一些研究[2, 3, 5, 6, 27, 28]使用該技術，該方法較易實作，且參數調整與結果上只需兩個參數(support 及 confidence)，結果易於表達，但是用於程式分析上有其缺點，原因是程式分析中需注意到事件發生之順序。

舉例來說，在 C 語言中 malloc 函式之使用必在 free 函式之前，否則記憶體配置會被破壞，而該方法恰缺少元件間前後關係的表達，在此例中 association analysis 可找出 malloc 及 free 函式之共同出現性，但無法說明 free 在 malloc 之前或之後。

2.5.2. SEQUENTIAL PATTERN

爲解決前後次序之問題，有另外的研究如[29]等，其方法根基於 association rule 上，進一步地解決此問題，不過目前尚未發現有論文將其應用於程式分析上。

舉例來說，假設有一程式，其內容如圖 7 所示，經編譯後執行，假設執行時可完整測試各個 execution path（即各種 condition），則結果可整理成表 3 之內容，其中第一欄 sequence 表示不同次的 execution path，而 order 表示當中的執行次序，在此以一次的 function 爲一單位，而最後的 Item 項則是該次 function call 中含有的 progam element，包含回傳值、函式名稱以及所使用之參數。

```
Enter function
if (condition_1) then
    A( );
    B( );
else if (condition_2) then
    C(D);
    A( );
    E = F(G );
else if (condition_3) then
    G = A(H);
else if (condition_4) then
    A();
    E = I(G);
    B();
else if (condition_5) then
    B();
Exit function
```

圖 7 Sequential Pattern 示範程式

表 3 轉換爲 Sequential Pattern 格式

sequence	Order	Item
1	1	A
1	2	B
2	1	C, D
2	2	A
2	3	E, F, G
3	1	A, G, H
4	1	A
4	2	E, I, G
4	3	B
5	1	B

資料整理結束後，設定閥值(threshold)以過濾發生頻率過低之 element，並再次轉換資料使各 element 對映至一集合(表 4)。再經過轉換，如表 5 所示。

表 4 element mapping

Large Itemsets	Mapped To
(A)	1
(E)	2
(G)	3
(E G)	4
(B)	5

表 5 Transformation in Sequential Pattern

Sequence id	Original Sequence	Transformed sequence	After Mapping
1	<(A) (B)>	<{(A)} {(B)}>	<{1} {5}>
2	<(C D) (A) (E F G)>	<{(A)} {(E) (G) (E G)}>	<{1} {2, 3, 4}>
3	<(A G H)>	<{(A) (G)})>	<{1, 3}>
4	<(A) (E G) (B)>	<{(A)} {(E) (G) (E G)} {(B)}>	<{1} {2, 3, 4} {5}>
5	<(B)>	<{(B)}>	<{5}>

再將此轉換後的 sequence，置入 candidate selection 的過程，假設 threshold 爲２，即 suport 值需大於二 。

Sequence	Support
<1>	4
<2>	2
<3>	2
<4>	2
<5>	3

Large 1-sequence

Sequence	Support
<1 2>	2
<1 3>	2
<1 4>	2
<1 5>	2
<2 3>	0
<2 4>	0
<2 5>	1
<3 4>	0
<3 5>	1
<4 5>	1

Large 2-sequence

Sequence	Support
<1 2 3>	0
<1 2 4>	0
<1 2 5>	1

Large 3-sequence

圖 8 candidate selection

最後的得出 sequence 爲<1 2>、<1 3>、<1 4>、<1 5>，對映回原本的表示法則是<(A) (E)>、<(A) (G)>、<(A) (E G)>、<(A) (B)>。其中因爲(E G)包含(E)且包含(G)，則<(A) (E G)>即可表示出<(A) (E)>、<(A) (G)>，故可削去。最後得<(A) (E G)>、<(A) (B)>，後者可解釋爲當使用函式呼叫 A 後常伴隨函式呼叫 B。

3. 系統架構與設計

本章討論本文之實作方式，本系統於 Linux 作業環境下發展，使用 CVS、CIL 以及 Python 等工具，在第一節內將整體描述本系統的三大部份，後續各節詳細討論每個部份的設計內涵以及實作方式。

3.1. 系統整體架構說明

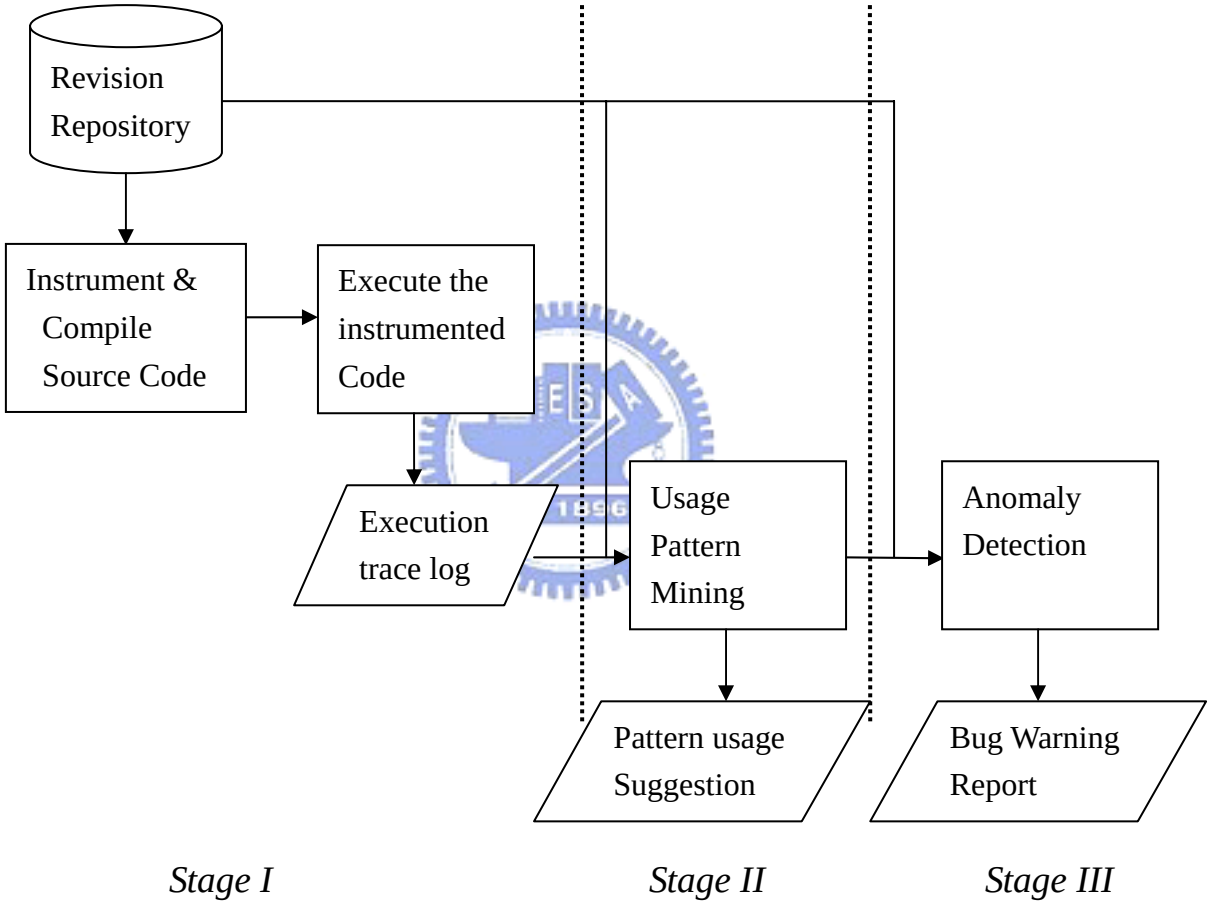


圖 9 系統架構圖

本篇論文的實作內容可分爲三個部份，第一階段是將檔案從版本庫(revision repository)中取出需要的檔案，加以適當處理(instrumentation)後，編譯(compile)並執行，最後取得程式執行歷程(log)。將此 log 置入第二階段之樣式探勘(pattern mining)，結果可以供使用者查詢特定函定之常見使用方法。第三階段是透過已找出的 pattern 以及專案發展過程(revision history)找出可能的錯誤情形，供程式開發者做爲除錯之參考。

3.2. 建立基礎資料

本階段主要目的在於建立實驗環境以供後續階段之使用，在此分爲四個部份解說。

3.2.1. 架立版本資料庫

由於開放原始碼運動近幾年來廣爲發展，有愈來愈多的專案公開於網路上，除了特定版本的原始碼公開外，更有許多專案將其開發歷程(revision history)完全公開，而這些歷程通常存於版本資料庫(revision history repository)中，目前開放原始碼常見的版本控制系統（Version Control System）如第二章相關研究所述，有 CVS(concurrent version system)[30]、SVN(subversion)[11]以及 git(使用於 linux kernel 發展)[31]等，每個專案有其不同的發展歷史、環境與成員，因此可能會使用不同的系統。

由於本研究需使用原始碼與版本資訊，並於其中進作大量的操作行爲，由於其資料庫幾乎都架設於國外，若使用外界版本資料庫做爲直接資料來源，將會使本研究所需之動作在執行時間上造成困難，因此本研究需在本地端重新複製一份資料庫，而 SVN 在資料庫複製上較 CVS 有其優勢，只要數行指令即可將遠端資料庫複製回本地，而 CVS 則不提供複製之功能，通常還需要其它工具的協助，如 CVSup[31]或 SVN-Mirror[32]。

基於以下的幾個原因，本研究以 OpenSSH[8]做爲分析標的：

1. 開發歷史長且完整。從 1999 年開始發展，至 2006 年六月廿二日止總計有 151307 次的更動紀錄，平均每週約有四百餘次更動發生；
2. OpenSSH 在程式安全上較被關注，因此有眾多資料可供參考並據以分析，另外也有相關的研究[4]是針對 OpenSSH 進行，可供本文分析之參考；
3. 提供 CVSup 之服務，在實驗環境架設上較爲便利。

OpenSSH 目前版本(4.3-p2)的總行數(Line Of Code, LOC)約爲十四萬行。

3.2.2. 修改與編譯原始碼

本階段的重點在於將原始碼中插入監測用的函式，用以紀錄程式中的部份語句(statement)。在第二章相關研究中曾討論數種修改(instrumentation)工具，各項工具有其侷限，其中如

1. GCC 雖便於使用，但是無法用以監控變數（如函式參數或回傳值）；
2. TXL 需針對程式中各種情況(scenario)而撰寫不同的轉換方式，當程式碼稍有變化即需要增加撰寫工作，故所需之作業甚爲繁複；
3. 使用商用軟體成本過高。

故本研究經比較後決定使用 CIL[25]做爲本階段使用之工具，並以 CIL 所內建之模組爲基礎，擴充其內容以符合本研究之需求，修改增加的功能如下段所述。

本研究注意的語句爲以下二種情形：

1. 語句中含有函式呼叫(function call)；
2. 承上之情形，和該函式呼叫相關之變數(variable)。

對於情形二所述變數，本文著重在同一語句內與函式呼叫有關的參數及函式回傳值(或稱 L-value)。

3.2.2.1. 演算法設計與實作

本演算法以 Ocaml 實作於 CIL 系統中，以模組化的方式寫成，以供日後方便使用，其設計如下：

```
While (not finishing traverse) {
    If (statement contains function_calls) {
        foreach function_calls:
            write2log("CALL %s", function_name)
            foreach argument in function:
                if argument is symbol:
                    write2log("ARGU %s", argument)
            write2log("DEP %s", return_value)
    }
    Next statement;
}
```

圖 10 instrumentation 演算法

以下舉實例說明：

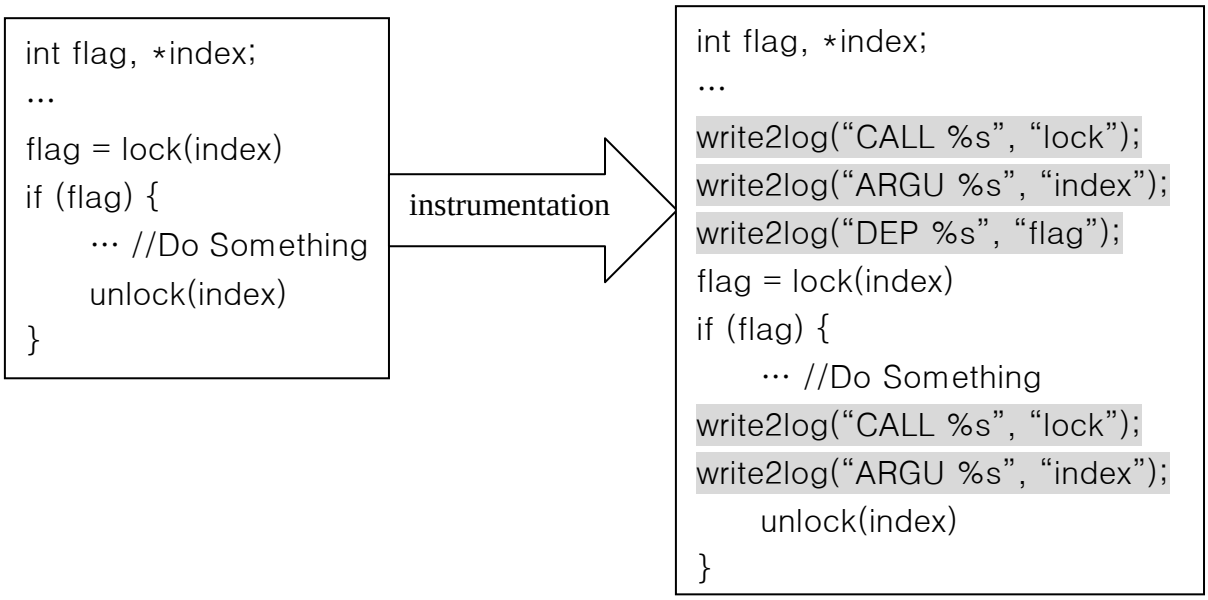


圖 11 instrumentation 實施例

如上圖所示， 研究者自行撰寫一函示（write2log，即圖中反白之部份）供監測使用，其功能在於記錄執行時期之訊息，並儲存於磁碟機等永久性儲存裝置上。目前本研究記錄的監測重點有三：

1. 函式呼叫－即上圖之 write2log("CALL %s",…)；
2. 和函式呼叫相關之參數－即上圖之 write2log("ARGU %s",…)；
3. 和函式呼叫相關之回傳值－即上圖之 write2log("DEP %s",…)；

3.2.2.2. 實作上所遭遇之困難

在實作上的困難有下列幾點：

1. 由於研究者自行撰寫之 write2log 函式，其內部使用 sprintf 函式，故會有 format string 之困擾，舉例來說，當 instrumentation 程式遇到 foo("PRINT THIS")時，照上述之演算法會轉換成：

```
write2log("CALL %s", "foo");
write2log("ARGU %s", ""PRINT THIS"" ); //invalid statement
foo("PRINT THIS");
```

顯而易見的是第二行爲不合法之語句，此情形將會造成 instrumentation 失敗，在此所提出的解決方法是不記錄某些型態的參數，例如含有雙引號或單引號之參數。

2. 在實作自行撰寫之 write2log 時，最早的做法是一有事件發生時即將內容記錄於檔案中，然而因爲寫入動作太頻繁，導致程式執行速度大爲降低，爲改善效能，之後改爲累積一定數量的事件後，再行寫入記錄檔，但這時卻造成另一個問題。

當程式結束前，留在記憶體(memory)中的 log 尚未累積到可寫入記錄檔的量，然而程式一結束時，記憶體中的資料即行消失，留存於記憶體中的事件

記錄也跟著消失，此時 log 檔便佚失最後的一次應寫入記錄。爲了解決此問題，研究者使用 atexit()函式，其工作原理是在程式執行中先規劃結束前所要執行的函式，當程式快結束時，即行呼叫事先所規劃之函式。我們在程式結束前透過 atexit()函式將記憶體中的事件記錄強制寫入磁碟，而後結束程式。

3.2.2.3. OPENSSH COMPILATION

本小節的目的在說明如何將以上發展的監測工具應用到實際專案中，這裡我們擇選以 OpenSSH 做爲分析的對象，在實作上有二個需克服的問題：

1. 由於研究者自行撰寫之監測函式，其內部也含有個多個函式呼叫，其呼叫是不在 OpenSSH 之正常執行序列中，爲避免監測程式監測其本身，故在 instrumentation 之前即將監測程式編譯(compiler)成目的檔(object file)格式，且不進行 Link。
2. OpenSSH 使用 GCC 中 attribute 的功能，但是有些語法並不是 CIL 所支援的，所以在 CIL 分析分程中會造成 CIL 錯誤。在專案中共有五處地方發生此類問題，在刪除這些 attribute 標示後，專案即可正常編譯。

3.2.3. 執行程式

使用動態分析需注意測試資料對整體程式碼的涵蓋率，惟目前並無針對整體程式的自動化測試工具，近年來發展的自動化測試工具，如 DART[33]，僅能針對單一函式做單元測試(unit test)，無法自動化進行整體程式測試，在衡諸現實情況後，使用 openssh 內建迴歸測試工具做爲測試資料的來源。

3.3. 程式常用樣式探勘

這一小節說明如何將前一階段得出的資料，經過轉換程序後，使用 sequential pattern mining 的方法找出常用樣式(frequent pattern)。

3.3.1. 資料整理

前一階段存下之 log 雖已記錄下所要的函式與變數名稱，但是這些內容往往夾帶過多的不必要資訊，其中最多的情況是變數的使用帶有 casting 之情形，舉例來說，foo(flag)以及 foo((int) flag)皆使用 flag 此變數做爲參數，但是後者多(int)此種 casting 寫法，實際上兩者的意義是相同的，但是 log 中前者記錄爲"flag"而後者記錄爲"(int)flag"，之後的分析工具會將兩者視爲不同的變數，但實際上是相同的，因此在此部份須將此種 casting 情況排除。

另外如果函式呼叫的參數爲固定數值或常數，則該參數在本研究中並無分析之必要。

3.3.2. 轉換結構

由於本研究是以 intra-procedure 做爲分析基礎，而 log 紀錄是以 inter-procedure 方式進行，因此需要適當轉換後方可進行接下來的工作。以圖 12 爲例說明，圖中左邊是原本的 execution trace，其中 A1 代表一次的函式呼叫，同樣地 A2~A4 及 B1~B3 各表示不同的函式呼叫，而 A3 函式中又呼叫了 B1 B2 B3 三函式，因此我們將 B1 B2 B3 此部份抽出，結果如右圖所示，分別兩不同之 sequence－ {A1 、A2、 A3、 A4} 及 {B1 、B2、 B3}，以此 sequence 爲單位進行下一步的分析

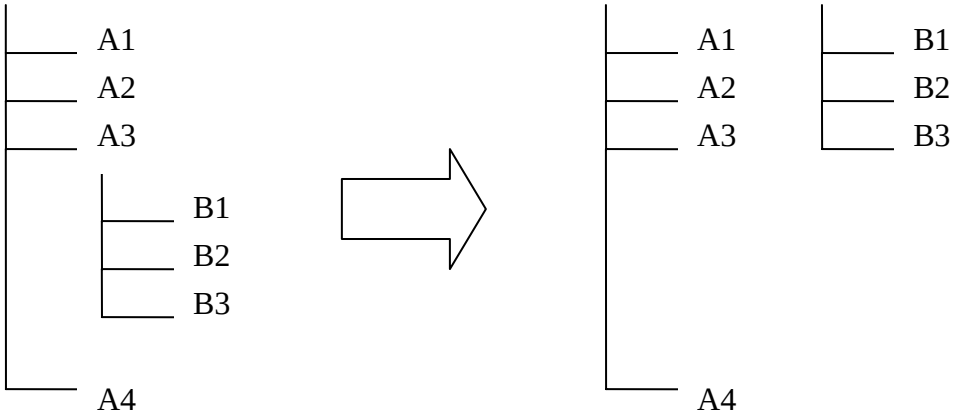


圖 12 TRACE 轉換示意

3.3.3. 變數更名

首先處理的問題是同樣功能的寫法被視爲不同 pattern 之情形。考慮以下兩種情況：

… sfd = socket(…); … bind(sfd, …); …	… sockfd = socket(…); … bind(sockfd, …); …

以上二種程式寫法具有相同之功能，但在寫法上略有不同，左邊的寫法使用 sfd 儲存資訊，而右邊則使用 sockfd，兩者的功能相同，但在使用 sequential pattern 之 mining 方法時可能會因爲名稱殊異而被視爲不同的 pattern，在上例中分別爲

<{sfd, socket} {bind, sfd}>　　<{sockfd, socket} {bind, sockfd}>

若在分析時能將名稱統一則能解決此問題，因此我們將該變數名稱改爲與函式名稱相關，因此結果分別爲：

<{ T-socket, socket} {bind, T-socket }>

<{ T-socket, socket} {bind, T-socket }>

二者在字序上完全相同，達到視此二者爲相同 pattern 之目的。

演算法如下：

```
Foreach statement in trace
    foreach argument in this line
        if argument was assigned in a previous function call
            Rename argument to the previous function-related name
    if this statement contains L-value assignment
        Rename this L-value to function-related name
```

圖 13 變數更名演算法

3.4. 錯誤警示

本階段之工作目標是取得在版本資料庫中特定型態之差異(diff)，在此我們將該型態鎖定爲只有一行之變動，之後再將此統計資料加入圖 9 系統架構圖之 stage II 所得 pattern。在實作上可分爲兩部份，一是處理版本資料庫並從中萃取資訊，二則是將此資訊連結至上一節之結果。

3.4.1. 整理版本資料庫

在 cvs 儲存方法是以檔案做爲版本之單位，意即在原始的設計上僅有檔案的版本號是可直接取得，其它如檔案內容－如函式或結構(structure)定義－須進一步分析才能界定是否在該檔案版本號下有所變動。各檔案內除了完整儲存最新一版之外，其它版本皆以差異(diff)的方式儲存，其檔案內容架構如下：

表 6 CVS 儲存格式說明

A. 此檔案的相關資訊－如版本之繼承或分支關係、版本之修改者與日期等
B. 基底版本(baseline revision)－即所紀錄之唯一完整版本，除明確表示其版本號外，其中又包含兩個 section
   i. Log－在 cvs 的變動行爲中，變動的發起者（submitter）可以文字敍述記錄相關資訊；
   ii. Text－完整的程式本文
C. 差異版本－記錄與下一版之間之差異，同樣地分爲兩部份
   i. Log－同上項
   ii. Text－以 diff 之方式儲存與下一版之間的差異。

在此一部份我們只需要上述之 C.ii 部份，使用 python 直接處理檔案內容，這些檔案包括版本資料庫中所有具有原始碼(source code)之檔案，使用適當的分節技巧後，略過不相關的 A、B、C.i.部份，將所有之差異寫入一個暫存檔中，作法如圖 14。

但是 diff 之內容可能已失去其完整性而難無法使用 CIL 等工具分析，以表 7 爲例說明，表中爲同一個檔案的二個版本，其中 Version 2 增加或修改部份內容，改變處有三，其一是註解增加一行，而我們也不需此一部份，但當 CIL 等程式分析工具處理包含此註解的全部差異時，由於此部份是以自然語言(natural language)表達，顯然會造成程式分析工具錯誤；其二是增加一函式呼叫 another(flag)，其爲完整之 statement，可直接由程式分析工具處理，但是考慮到例中第三種情況時，由於 C 語言容許一個 statement 在多行完成，而 cvs 是以一行爲單位做 diff，則此情況會造成衝突，此例中 buk 此函式呼叫更動一參數，而該參數又恰巧位於下一行中，只使用 diff 之結果無法回建 AST，導致 CIL 無法使用。

故若在處理 DIFF 時使用較爲正規的方法（如 CIL）會導致研究難以進行，因此選擇針對 openssh 之情況，研究者以程式自動化地判斷各 diff 是否爲所需資料，詳細內容

及演算法如圖 15 圖 15 所示。

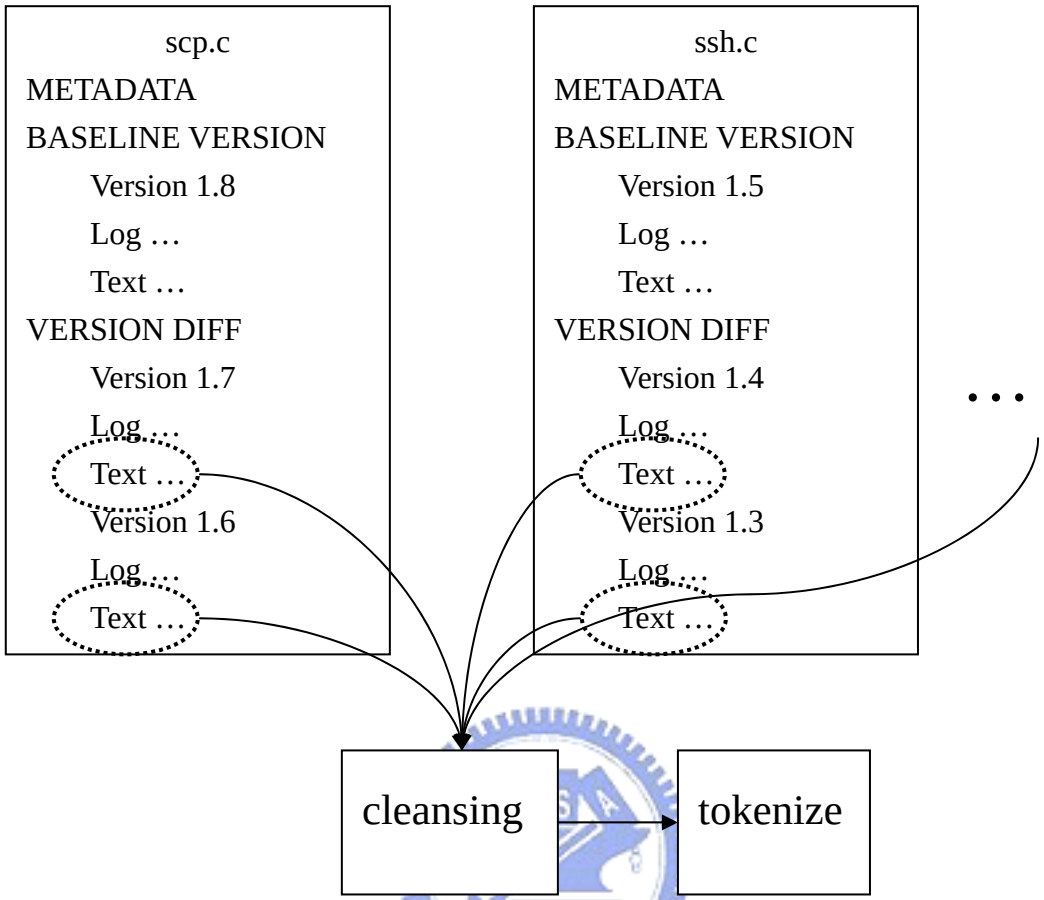


圖 14 diff 處理示意

表 7 版本差異示例

Version 1	Version 2	Diff(version1, version2)

int foo(void)
{
/*
Comment line 1
*/
int i;
i = buk(“SHOW”,
1);
return i;
}
	int foo(void)
	{
	/*
	Comment line 1
	Comment line 2
	*/
	int i;
	another(flag);
	i = buk(“SHOW”,
	2);
	return i;
	}
		add 4 1
		Comment line 2
		add 6 1:
		another(flag);
		del 8 1
		add 8 1
		2);

虛擬碼	說明
foreach file in repository skip to diff section;	直接讀出版本庫中每個檔案的 diff section(表 6 之 C.ii 部份)
Foreach diff Foreach line in the diff	對各 diff 中的每一行
if (this line is comment) or	該行是註解，除了正常的註解方式，如//或/* */，openssh 關於註解的特別處是其在多行註解中（如表 7 中之 Comment line 2）雖無//或/* */，通常以星號*做爲該行起始，在排除掉此星號是作指標(pointer)用途後，即可認定此行爲註解
(this line starts with RCSID) or	早期的 Openssh 變動常會在有效程式碼中加入 RCSID()函式，此函式目的在於記錄目前的版本，不影響程式的作爲，因此可忽略
(this line is preprocessing) or	該行以#include 或是#ifdef 等前置處理單元
(this line contains no alphabet)	該行不含任何字母
Pruning this line	若符合上列之各種情況，在該 diff 中忽略此行
if this diff contains exactly only one line tokenize this line foreach token variant[token]++	當該 diff 中我們所關注的實際內容，其行數爲一行時，紀錄該行中每一個 token 之數量

圖 15 diff 處理演算法

3.4.2. 對映至 MINED PATTERN

將上一小節之結果－即各 token 之數量－對映到各 stage II 所得之 pattern 中，toekn 可能爲 pattern 內之函式或變數，我們會在第四章中討論 pattern 內各 token 之關係以及 pattern 間如何利用 token 衡量其重要性。

4. 實驗結果

4.1. 函式呼叫內容

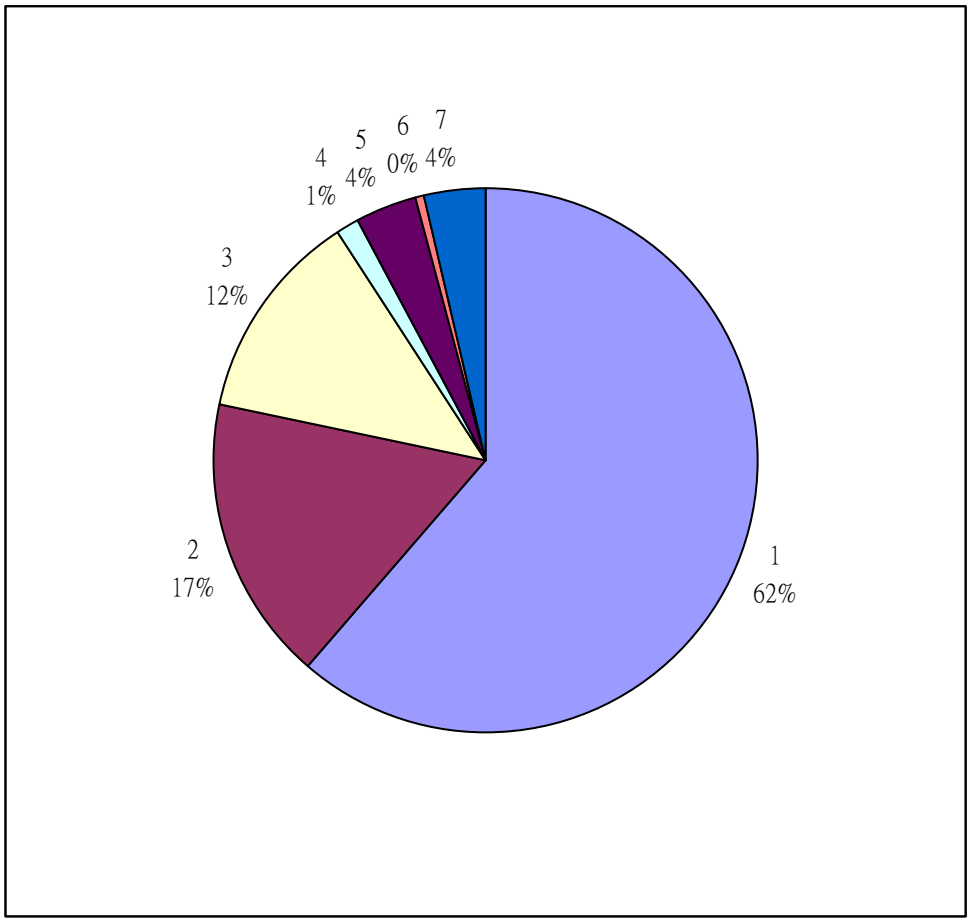


圖 16 函式呼叫長度

在 4089 次的函式呼叫中，62%的函式內部僅含有一次函式呼叫,這些函式在 intra-procedure 的方法中無法產生 pattern，因此在之後的分析中我們將去不考慮此情形。另外，僅有４%的函式呼叫其長度是在七次或七次以上，而長度大於６４僅有５次。

4.2. SUPPORT 與 RULE 數量

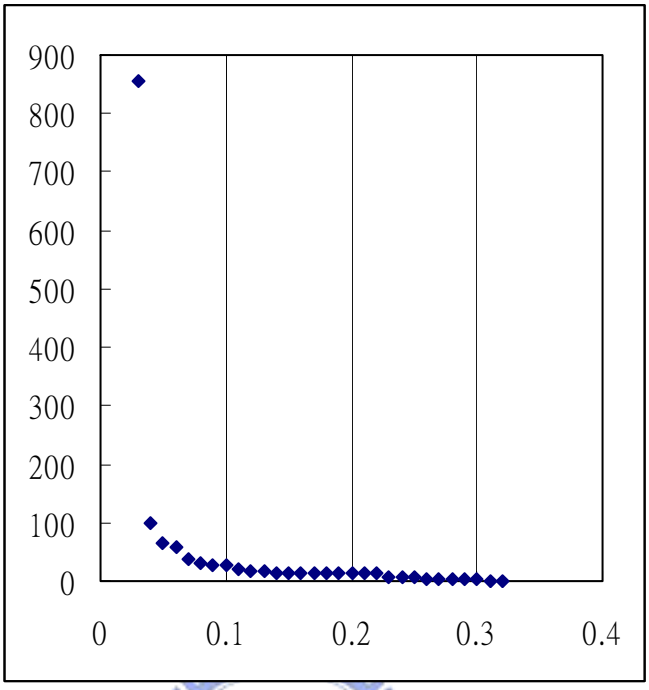


圖 17 SUPPORT 與 RULE 數量

圖中橫座標爲 threshold 值，縱座標爲在各以 threshold 值爲底限所產出的 rule 數量，其中各點爲獨立之實驗。

在資料探勘中 support 值決定 rule 產生之數量，在這份研究中在 support 值設爲 0.03 時（即至少出現 1576*0.03=47 次），rule 約產生 850 則，而 0.04 時則降至 100 則，至 0.33 時則無 rule 產生。高的 support 雖然較準確，但會造成可能忽略次數低於閥值(threshold)的 rule，即 false-negative 之情形，而相反地，低 support 雖較能找出 rule，卻可能造成 false-positive 之情形。

4.3. RULE 長度

這裡將 rule 長度定義爲 rule 中所包含的集合數量，而一個集合包含一函式及零個以上的變數。舉例來說，{channel_prepare_select}→{packet_have_data_to_write}→{select}爲所找出的 rule，代表三個程式實體接連被呼叫或使用，則此 rule 長度爲３。

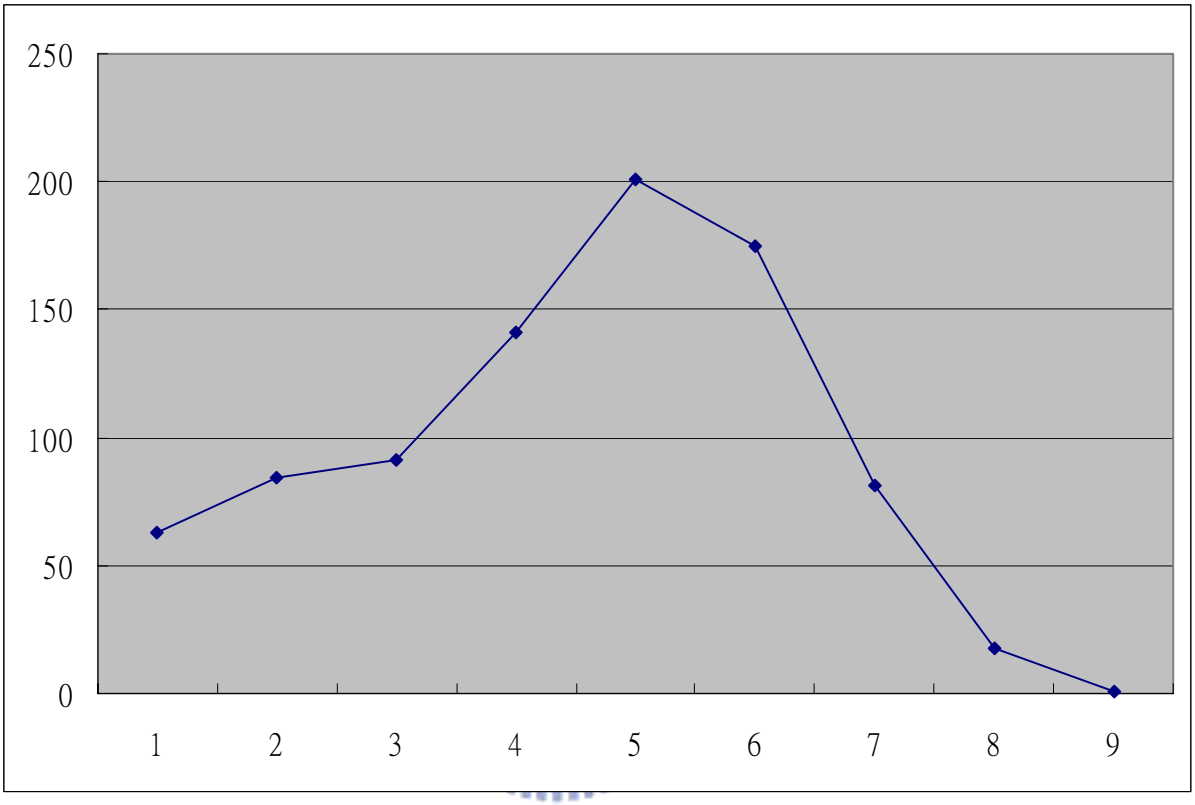


圖 18 rule 長度統計

圖中橫座標爲 rule 長度，縱座標是該 rule 長度的 pattern 之數量，threshold 爲至少出現 47 次。

此圖中說明我們的設計中，所找出的 rule 最長可爲八，而有 90%以上的 rule 其長度介於二到五之間，而長度爲一的 rule 雖無法用於撰寫建議，但可以做爲錯誤偵測之後，其使用方法見下節。

4.4. RULE 說明

No	Support			
i	101	c->input		
ii	95	c->output		
iii	103	c->extended		
iv	50	c->input ➔	c->output	
v	50	c->input ➔	c->output ➔	c->extended
vi	100	c->input ➔	c->extended	
vii	53	c->output ➔	c->extended	

上表比較 c->input、c->output 以及 c->extend 三個變數之關係，最左欄表示此 rule 之出現次數，舉第 v 項 rule 說明，其表示程式執行中曾發生過 c->input、c->output 以及 c->extend 曾在某一函式中有次序地被使用。

若比較 iv 項第與第 v 項 rule，則可得出一結論：

若已知 c->input、c->output 兩變數接連使用，則 c->extended 隨後也將被使用。

使用貝氏機率表示其可能性 Prob(support(v)|support(iv))＝１００％，即 confidence 為１００％，表示其 pattern 為可靠的。

故當程式中已使用 c->input 及 c->output 時, 若 c->extended 不被使用則可判定為程式錯誤

另外，若比較第 i 項及第 vi 項，則可得以下之結論：

若已知 c->input 變數被使用，則 c->extended 隨後也將被使用。

其 confidence 為９９％，表示其 pattern 可能為可靠的。

程式中只有一處不符此 pattern 之情形，此處可判定為錯誤

比較(iii, vi)以及(v, vii)可樣可得此種結果

另外，若比較第 i 項及第 iii 項，則可得以下之結論：

若已知 c->input 變數被使用，則 c->out 隨後也將被使用。

其 confidence 為５０％，表示其 pattern 為不可靠的。

程式中有多處地方不合此 pattern，此處可判定此 pattern 為錯誤

由上述之三種情況可將得出的 rule 分爲 frequent、Potential-Error 及 Unlikely 三種，決定標準在於 confidence。以上討論的三個變數在版本歷程中不曾出現一行變動(one-line checkins)的情形，下一頁的示例討論加入一行變動之情形。

No	Support			
a	58	s = strlen(str)		
a.l		18 1		
b	66	strlcpy (str, len, m)		
b.l		4 1 74		
c	60	m = xmalloc(len)		
c.l		22 74		
d	58	s = strlen(str) ➔	strlcpy (str, len, m)	
e	58	s = strlen(str) ➔	m = xmalloc(len)	
f	58	m = xmalloc(len) ➔	strlcpy (str, len, m)	
g	58	s = strlen(str) ➔	m = xmalloc(len) ➔	strlcpy (str, len, m)
g.l		18	22	4

上表比較 strlen、strlcpy 及 xmalloc 三函式及其相關變數之關係，其中 row a.1 對映至 row a 之項，該數字表示對映的辯識符(identifier)曾在一行變動(one-line check-ins)中被更動的次數。由於變數名稱較函式名稱變動容易，因此統計變數的變動次數可能較無意義。

變動次數用於文件製作的方式有以下二種：

- 當使用者查詢某一函式的用法時，結果出現多種 pattern，則 pattern 可按照以下二參數排序：confidence 以及函式變異程度平均值，假設 rule g 為其中之一，則其函式變異程度平均值為(18+22+4)/3=15
- 描述 pattern 時可以用變動次數標示（highlight）重要的部份，如 rule g 中可特別標示 xmalloc，以表示這部份最常出錯，防止使用者在撰寫新功能時錯誤或是在除錯(debug)時可特別注意此處。

4.5. 例外情況

在 Socket programming 中，通常一個 client 程式其建立 tcp socket connection 的動作只會出現一次，因此在 sequential pattern 分析的初始即會因發生次數(support)不足而被捨棄。

另外在 openssh 中大量使用記憶體相關的函式，因此造成分析時母體數過高，造成其它發生次較低的 pattern 被忽略。

5. 結論

本論文使用 sequential pattern mining 方法分析 openssh 程式之動態歷程，輔以在 openssh 版本歷程中的變動情況，以描述程式元素(program element)之關係，其中使用 Support 用以挑選出現次數夠高的 pattern，接著以 Confidence 決定 pattern 之品質，其品質可分爲三類:

- frequent： 該 pattern 於執行中完全無例外情況，可做爲使用建議
- Potential Error：該 pattern 於執行中有少數例外情況，可檢視該例外情況以除錯
- Unlikely： 該 pattern 於執行中有許多例外情況，pattern 可能實際上不存在

最後以 Variant 比較不同 pattern 的重要性，以及 pattern 中各單元的重要程度。

本研究有二個主要貢獻，其一是相較其它論文所使用的 apriori-based 之方法，使用 sequential pattern 方法，可以確定 pattern 中先元素的前後次序；二是導入變動情況的概念，故可以更明確地描述 pattern，並顯示出易錯誤處。

5.1. 與相關研究之比較

	分析實體	分析範圍	特點比較
Guide Software Changes (2004)		Each revision diff	• Less dedicated program analysis
System specific Rule (2005)	Procedure call	10 lines in a revision	• Pairwise pattern • Static analysis
Matching Method calls (2005)	Procedure call	Each revision diff	• Pairwise pattern • Dynamic Analysis
PR-Miner (2005)	Procedure call and variable	intra-procedure	• Static analysis
Our work	Procedure call and variable	intra-procedure	• Dynamic analysis • Ordered pattern

5.2. 未來改進方向

5.2.1. 偵測異名變數

C 語言由於其指標(pointer)便於使用，因而功能強大，然而指標的使用卻帶來程式分析時相當大的困擾。以下列三行 C 程式爲例說明：

```
int * flag1 = 3;
int * flag2;
flag2 = flag1;
```

在執行後，記憶體內的有一塊單元其值爲三，而 flag1 及 flag2 皆指向此位址，使用指標透過 flag1 修改該單元的值也會造成 flag2 指向的值改變，反之亦然，因此在此 flag1 及 flag2 具有相同之意義，然而在本論文中 flag1 及 flag2 是視爲不同的，例如:

```
foo(flag1);
```

以及

```
foo(flag２);
```

將會被統計爲不同的 pattern，由於該問題爲 NP-Complete program，因此若加入 pointer alias analysis 之近似演算法應可以部份解決此問題。以上的問題可能造成 false-negative 之情形（即存在的規則被忽略而沒被找出）。

5.2.2. 改善版本變動歸因

由於本研究在處理版本資料時皆只使用一行修正(one-line check-ins)，若有一組重要的函式使用方式是從未出錯或恰好其修改皆在二行以上，則我們的研究將無法彰顯這些函式的重要性。若能有更細緻的演算法標定每次的版本更動的性質，則能更精確描述每個 pattern 之變動性或重要性。

5.2.3. 利用資料相依性

本研究僅使用變數更名的方式以表達資料相依性(data dependency)，然而此種簡易的方法仍不足我們的需求，若能透過變數展現函式間的關係，搭配前文之 sequential mining 之方法，則有助於找出低發生次數的 pattern，並降低 false-pattern 的機會

參考文獻

[1] Y. Shigio, "GNU GLOBAL source code tag system",http://www.gnu.org/software/global/

[2] Z. Li and Y. Zhou, "PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* 2005, pp. 306-315.

[3] B. Livshits and T. Zimmermann , "Locating matching method calls by mining revision history data," in *Workshop on the Evaluation of Software Defect Detection Tools,* 2005.

[4] I. Neamtiu, J. S. Foster and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories,* 2005.

[5] C. C. Williams and J. K. Hollingsworth, "Recovering system specific rules from software repositories," in *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories,* 2005.

[6] T. Zimmermann, P. Weisgerber, S. Diehl and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering,* 2004, pp. 563-572.

[7] R. Purushothaman and D. Perry, "Towards understanding the rhetoric of small changes," in 2004, pp. 90-94.

[8] "OpenSSH" http://www.openssh.com/, May, 2006.

[9] D. Kramer, "API documentation from source code comments: A case study of javadoc," in *SIGDOC '99: Proceedings of the 17th Annual International Conference on Computer Documentation,* 1999, pp. 147-153.

[10] A. Zeller, "Configuration Management with Version Sets," *Abteilung Softwaretechnologie, Technische Universität Braunschweig, Braunschweig,* 1997.

[11] CollabNet, "subversion", http://subversion.tigris.org/

[12] S. Huang and K. Liu, "Mining version histories to verify the learning process of legitimate peripheral participants," in *MSR '05: Proceedings of the 2005 International Workshop on*

Mining Software Repositories, 2005.

[13] D. Cubranic, Murphy and Gail C., "Hipikat: Recommending pertinent software development artifacts," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering,* 2003, pp. 408-418.

[14] T. Zimmermann and P. Weisserber, "Preprocessing CVS data for fine-grained analysis," in *MSR 2004: International Workshop on Mining Software Repositories,* 2004.

[15] I. Sommerville, *Software Engineering.* ,7th ed.Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2004,

[16] J. Whaley, M. C. Martin and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis,* 2002, pp. 218-228.

[17] S. Konrad and B.H.C. Cheng, "*Requirements Patterns for Embedded Systems,*" *IEEE Joint International Conference on Requirements Engineering, 2002. Proceedings., 2002, pp.127-136*

[18] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," *.Proceedings.International Conference on Software Maintenance, 2003.,* pp. 305-314, 2003.

[19] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," *ACM SIGSOFT Software Engineering Notes,* vol. 27, pp. 229-239, 2002.

[20] R. Kollmann, P. Selonen, E. Stroulia, T. Systa and A. Zundorf, "A study on the current state of the art in tool-supported UML-based static reverse engineering," *Proceedings.Ninth Working Conference on Reverse Engineering, 2002.,* pp. 22-32, 2002.

[21] L. C. Briand, Y. Labiche and Y. Miao, "Towards the reverse engineering of UML sequence diagrams," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering,* 2003, pp. 57.

[22] A. Zeller and D. Lutkehaus, "DDD—a free graphical front-end for UNIX debuggers," *SIGPLAN .,* vol. 31, pp. 22-27, 1996.

[23] B. Demsky and M. Rinard, "Data structure repair using goal-directed reasoning," *Proceedings of the 27th International Conference on Software Engineering,* pp. 176-185, 2005.

[24] J. Cordy, "TXL-A Language for Programming Language Tools and Applications,"

Proc.4th Int.Workshop on Language Descriptions, Tools and Applications, Electronic Notes in Theoretical Computer Science, vol. 110, pp. 3 – 31, 2004.

[25] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," *Conference on Compiler Construction,* 2002.

[26] J. Han and M. Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufmann, 2000,

[27] A. Michail, "Data mining library reuse patterns using generalized association rules," *International Conference on Software Engineering,* pp. 167 – 176, 2000.

[28] A. Michail, "Data mining library reuse patterns in user-selected applications," *Automated Software Engineering, 1999.14th IEEE International Conference on.,* pp. 24-33, 1999.

[29] H. Mannila, H. Toivonen and A. I. Verkamo, "Discovering frequent episodes in sequences," *KDD,* pp. 210-215, 1995.

[30] "CVS - open source version control," http://www.nongnu.org/cvs/

[31] John Polstra, "CVSup," http://www.cvsup.org/

[32] K. CL, "SVN-Mirror," http://search.cpan.org/~clkao/SVN-Mirror-0.68/

[33] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation,* 2005, pp. 213-223.